C++

MS-DOS COMPILER

ZORTECH

ZORTECH C++ COMPILER

The World's First 'TRUE' C++ Compiler for MS-DOS Machines

Marketed in the USA by:

Zortech Inc, 366 Massachusetts Avenue, Arlington, MA02174

Tel: 617-646-6703 fax:617-643-7969

and in the rest of the World by:

UK Office : Zortech Limited, 106-108 Powis Street, LONDON, SE18 6LU

Tel: 01-316-7777 fax: 01-316-4138

COPYRIGHT 1986, 1987,1988 - ZORTECH LIMITED

ZORTECH C++ COMPILER

The World's First 'TRUE' C++ Compiler for MS-DOS Machines

THE ZORTECH C++ LICENSE AGREEMENT

1. Please do not give away copies of this product to your friends - encourage them to buy a legitimate copy for themselves. This is the only way that the programmers, writers and employees of Zortech can be rewarded for the incredible amount of work that has gone into this product.

2. Please do not load the software onto more than one machine at your place of work - this is illegal and also cheats the publishers and programmers out of their rightful income.

3. Treat this software like a book - two people cannot easily read the same book at the same time, can they? Each individual who wants to read a book needs to purchase his own individual copy - likewise, if you are using this product please make sure you have purchase a legitimate copy.

4. Zortech Ltd, it's subsidiary companies, distributors, dealers and employees cannot be held responsible for any financial loss incurred or otherwise, caused by any failure or weakness of this product.

COPYRIGHT 1987, 1988 ZORTECH LIMITED
SECOND PRINTING

With Special Thanks To:

Walter Bright *- for giving us all something EXCITING to work with.*
Bruce Eckel *- for helping us out with the C++ TEXT and beta-testing.*
Bjorn Freeman-Benson *- for his super work on the Zortech LINKER.*
Steve Teale *- for his invaluable service on the Zortech HOTLINE.*
Joe Huffman *- for writing the fastest GRAPHICS library ever!*
Nikki Locke *- for providing us with the ZED environment.*
Kevin Powis *- for the context sensitive HELP system.*
Dave Morton *- for getting the MANUAL started.*
Mark Welsh *- for his help in TYPESETTING the manual*
Dave Mansell *- for his TECHNICAL excellence and long hours.*
Tanj Bennett *- thanks for the LIBRARIAN - we still use it!*

and a very special mention to:

Bjarne Stroustrup *- thanks for THE BOOK!*

PREFACE

Welcome to Zortech C++!

This is the first native-code C++ compiler for the IBM PC/XT/AT class of machines. "Native code" means you don't need to spend a lot of money on someone else's C compiler. In fact, you've just purchased a C++ compiler, a C compiler, a linker, a librarian and a powerful make utility. You can mix and match code in C++, C and assembler. We've included a dynamite editor (ZED) which runs the compiler for you, stops if an error occurs, and positions the cursor for easy correction.

Zortech C++ comes with a complete C++ and C library, including support for mice, fast screen writes, BIOS / DOS Calls and interrupt handling. You can use Microsoft's Codeview for run-time debugging. Altogether, it's the most powerful, inexpensive object-oriented development system available for the PC.

C++ was developed at Bell Labs; it's an object-oriented extension to the C language. You can easily compile your existing C code and start using the extensions as you learn them. In addition, C++ allows improved error checking, which helps find bugs faster. Since C++ is object-oriented, you can create your own data types. This can make the solutions to your problems much cleaner. For instance, if you're developing a weather-mapping system, create a data type called `StormFront`, declare a bunch of `StormFronts` and move them around, combine them, make them rain, etc. This is higher-level thinking, and you can solve more complex problems.

C++ helps you easily re-use old code (even if they are object files without the source). If you're an astronomer and someone else has defined a data type called a `MassiveBody` which isn't *quite* what you want, you can say: an `Asteroid` is a `MassiveBody` with a few changes and additions.

You don't even need to understand *how* a `MassiveBody` works -- just *what* it does. Since you only need to write and debug the changes and additions, you get results faster.

When using C++, you'll find that the structure of the language practically organizes your code for you. Start by building some of the pieces of your project (`StormFronts`, `Asteroids`). Once these are tested, it's a snap to integrate them in with the rest of the system (and the compiler generates errors if you try to do something a type wasn't designed for).

The goal of C++ is to help you program faster. It doesn't come at the expense of efficiency, though - you'll find your C++ programs are generally as fast and small as your C programs.

Table of Contents

Part One - User Guide

QUICK START ***9***

Quick Installation 9

Quick Editing 10

INTRODUCTION TO C++ ***13***

Object-Oriented Programming 14

Efficiency 15

Quick Pre-requisites 16

Declarations vs. Definitions 16

Function Prototyping 16

Using Header Files to Declare Classes 17

Quick Introduction to Streams 18

Classes - Defining Your Own Types 19

A Complete Example 21

Function Overloading 24

Operator Overloading 28

Making a New Type From an Old Type 31

More Help is on the Way 35

SOPHISTICATED C++ ***37***

Calling Member Functions 37

Constants 37

Comments 38

Multiple Inclusion of Header Files 38

Scoping 39

Inline Functions 40

Access Functions 41

Dynamic Memory Allocation 42

Constructors & Destructors 43
References 45
Operator Overloading 49
operator=() 49
Void Pointers 51
Virtual Functions 52
Friends 55
Public, Private and Protected 55
Static Variables 56
The Scope Resolution Operator 56

Part Two - Reference Guide

DETAILED INSTALLATION GUIDE **59**
System Requirements 59
Hard Disk Installation 59
Environment Variables 60
Altering Directory Paths and Environment Variables 60
CFLAGS Environment Variable 62
Floppy Disk Installation 62

CONTEXT SENSITIVE HELP **63**
Introduction 63
Configuring the Software 63
Installing the Software 64
Loading the Resident Helper 65
Removing the Resident Helper 65
Invoking Help 65
Keys Summary: 66

ZED EDITING ENVIRONMENT **67**
Starting ZED 68
Keys Bindings 70
Cursor Keys 72

Wordstar Compatible Keys 74
Other Control Keys 75
Alternate Keys 76
Online Help 79
Menus 80
Main Menu 81
Files Menu 82
  Entering file names 85
  Size menu 87
Moves Menu 88
Edit Menu 90
  Size menu : 93
Blocks Menu 94
Delete Menu 96
Search Menu 97
  Entering strings 99
Other Menu 101
  Entering strings 102
Compiling 104
Compiling Projects With Multiple Source Files 107
Testing & Running DOS Programs 107
MSDOS Problems 109
Record and Playback Menus 110
Text Menu 111
  Formatting and Autoindent 112
Backups 113
Menu Structure 114
Configuration 115
  Set up Quick Command Keys (Hotkeys) 116
  Path Settings (Files) 122
  Screen Display (Screen) 123
  Screen colors (colors and More colors) 126
  Screen Layout (Lines) 127
  Window Positions 128
  Boxes 129
  Printing Configuration (Print) 129
  Saving Configuration (saVe) and Exiting (eXit) 130

ZORTECH COMPILER **131**
Overview of the Compilation Process 131
ZTC1.EXE, ZTC1B.EXE and ZTCPP1.EXE 133
Command Line....133
infile 133
switches 134
Environment Variables....139
ZTG.EXE....140
Command Line 140
Other Switches....141
ZTC2.EXE and ZTC2B.EXE 142
infile....142
switches 142
ZTC Command 143
Environment Variables....145

C++ REFERENCE **147**
Comments 147
Identifiers 147
Keywords 147
Conflicts With Identifiers in the Library 147
Character Constants....149
Types....150
const and volatile....151
extern and Global Variables 152
Alignment....153
Function Parameters 153
Pointers 154
Initializers....154
Floating Point Operations 155
The Preprocessor 156
Macros 156
Predefined Macros 157
Include Files 158
Other Preprocessor Directives....159
Limitations....160
Function Prototyping 161
Prototype Induced Cast 163
Prototype - char and float....163
Autoprototyping 164

Compatibility With C 166
User-defined Type Conversions 167
Compatibility with ATT C++ 167
C++ Programming Caveats, Bugs, Warnings 168
Mixing C and C++ Modules 170
Miscellaneous 171

LINKING PROGRAMS ***173***
LINK.EXE 173
Using with ZTC 174
Usage 174
switches 174
Environment Variables 176
Differences between LINK and MS-LINK 177
BUNCH.EXE 178
Using with ZTC 178
Usage 178
.COM files 178
Environment Variables 179
Bugs 179
EXE2BIN.COM 179
Integer Only Programs 180
Creating .COM Programs 180

PROGRAM EXECUTION ***181***
Command Line Argument Expansion 182
Controlling Stack Size 183

GLOBAL OPTIMIZER ***185***
Introduction 185
What is a Global Optimizer? 185
What a Global Optimizer will not do 186
Operation 187
Running the Optimizer with ZTC 187
Running Stand-Alone 187
Advanced Information 189
Constant propagation (+cnp) 189
Copy propagation (+cp) 190

Dead assignment elimination (+da) 191
Dead code elimination (+dc) 191
Global common subexpressions (+gcse) 192
Loop invariant removal (+li) 192
Loop induction variables (+liv) 193
Iterate until no more optimizations (+loop) 193
Do our own register assignments (+reg) 193
Space/time tradeoffs (+space and +time) 194
Very busy expressions (+vbe) 195
Hints and Tips 196
Use of const and volatile keywords: 196
If problems occur 197

ADVANCED FEATURES ***199***
Memory Models 199
8088/86 Segmented Architecture 199
Five Different Models 200
Choosing a Memory Model 200
Pointers 201
Near Pointers 201
Far Pointers 201
Macros for Use with far Pointers 202
Normalizing Far Pointers 203
Far Pointer Arithmetic 204
Using near and far Keywords 206
Using Near Pointers 206
Using Far Pointers 207
Converting Between Near and Far Pointers 207
Alignment Suppression Within Structures 208
Debugging C++ Programs 209
Using Debuggers 209
Other Aids 209
Debugging with CodeView 210

TOOLS ***211***
MAKE 211
What is MAKE? 211
Getting Started With MAKE 212

The MAKEFILE 212
An Example MAKEFILE 212
Dependency files 213
A MAKEFILE with More than One Source File 214
Running MAKE 215
Uses of the MAKE Arguments 216
User Defined Macros 217
Pre-Defined Macros 218
Rule Flags 218
MS-DOS internal commands 219
Special targets 219
Logging 219
Implicit Rules 220
Creating Linker Response Files 221
Summary of the differences from the UNIX MAKE 222
TOUCH 223
OBJTOASM 224
Usage 224

INTERFACING WITH ASSEMBLER **225**
Program Sections 225
T Memory Model (.COM Programs) 226
S Memory Model 226
M Memory Model 227
C Memory Model 227
L Memory Model 228
General Layout 231
Subroutine Linkage 232
Stack Frame 232
Function Return Values 233
Register Usage 233
Data Alignment 233
Macros in MACROS.ASM 234
Simple Example for S Model 235
Running MASM 242

Part Three - Function Library

THE FUNCTION LIBRARY **245**
File I/O 246
File modes 246
High and Low level file I/O 247
Memory Allocation 248
HEAP 248
STACK 249
Global Variables 250
_8087 250
_okbigbuf 255
_osmajor 255
_osminor 256
_psp 256
errno 256

ERROR MESSAGES **257**
DOS Errors 257
Math Errors 258

HEADER FILES **259**
assert.h 259
conio.h 259
ctype.h 259
direct.h 260
disp.h 260
dos.h 261
fg.h 262
errno.h 264
int.h 264
io.h 264
limits.h 264
math.h 264
msmouse.h 265
process.h 265

setjmp.h 265
signal.h 265
sound.h 265
stdarg.h 266
stddef.h 266
stdio.h 266
stdlib.h 267
string.h 267
sys\stat.h 267
time.h 267

LIBRARY FUNCTIONS **268**
abort 270
abs 271
acos 272
asctime 273
asin 275
assert 276
atan, atan2 277
atof, atoi, atol 278
bdos, bdosx 279
bioskey _bios_keybrd 280
bsearch 282
calloc 284
ceil 286
chdir 287
_chkstack 288
clearerr 289
clock 290
close 291
cos, cosh 292
creat 293
ctime 295
difftime 296
Display Package 297
div 303
DOS Package 304
ecvt 307
execl - execvp 308

exit, _exit 310
exp 312
fabs 313
farcalloc farrealloc 314
_farptr_norm _farptr_fromlong _farptr_tolong 316
fclose 318
fcvt 319
feof 320
ferror 321
fflush 322
fgetc 323
fgets 324
fileno 325
filesize 326
findfirst, findnext 327
floor 329
flushall 330
fmod 331
fopen 332
FP_OFF, FP_SEG 334
fprintf 335
fputc 336
fputs 337
fread 338
free 339
freopen 340
frexp 341
fscanf 342
fseek 343
fstat 345
ftell 347
fwrite 348
getc, getchar, getche, getch 349
getcwd 351
getDS 352
getenv 353
gets 354
hypot 355
index 356
inp, inpw 357

Interrupt Package 359
int86, int86x 362
intdos, intdosx 365
isatty 367
is package 368
itoa 371
kbhit 373
labs 374
ldexp 375
ldiv 376
localtime 378
log, log10 380
longjmp 381
lseek 383
malloc 385
matherr 386
memchr - memset 388
mkdir 390
MK_FP 391
mktime 392
modf 393
Mouse Package 394
open 396
outp, outpw 398
peek 400
perror 402
poke 403
poly 405
pow 406
printf, fprintf, sprintf 407
putc, putchar 412
puts 413
qsort 414
raise 416
rand 418
read 419
realloc 421
rename 423
rewind 424
rmdir 425

sbrk 426
scanf 428
segread 432
setbuf 433
setjmp 434
setvbuf 435
signal 438
sin, sinh 440
Sound Package 441
spawn 443
sprintf 445
sqrt 446
srand 447
sscanf 448
stat 449
strcat - strdup 451
strerror 453
strlen 454
strlwr 455
strncat - strpbrk 456
strrchr - strset 458
strspn 459
strstr 460
strtod - strtoul 461
strupr 464
swab 465
system 466
tan, tanh 468
time 469
toascii, tolower, toupper 470
ungetc 472
unlink 474
utime 475
va_arg - va_start 477
vfprintf, vprintf, vsprintf 478
write 480

FLASH GRAPHICS INTRODUCTION ***483***
Definition of Basic Coordinate System 484
Variable Types Defined 484
Constants Defined by FG 485
Graphics Display Adapters 485
Box and Line Indexes 486
Masks 487
Writing Modes 488
Types Of Lines 488
Available Colors 489

PUBLIC VARIABLES ***491***
fg_displaybox 491
fg_charbox 492
fg_ncolormap 492
fg_nsimulcolor 493
fg_pixelx, fg_pixely 493
fg_numpages 494
fg_display 494
fg_activepage 495
fg_displaypage 495

OPTIONAL USER-SUPPLIED FUNCTIONS ***497***
_assert 497
fg_lineclip 498

FLASH GRAPHICS PUBLIC FUNCTIONS ***499***
fg_adjustxy 499
fg_blit 501
fg_box_cpy 502
fg_drawarc 503
fg_drawbox 504
fg_drawdot 505
fg_drawellipse 506
fg_drawline fg_drawlinep 507
fg_drawlineclip 508
fg_drawmatrix 509

fg_drawthickline 511
fg_fillbox 512
fg_flush 513
fg_getcolormap 514
fg_init 515
fg_init_null - fg_init_vga12 516
fg_line_cpy 518
fg_putc 519
fg_puts 520
fg_readbox 521
fg_readdot 523
fg_restore 524
fg_save 525
fg_setactivepage 526
fg_setcolormap 527
fg_setdisplaypage 528
fg_setlinepattern 529
fg_term 530
fg_writebox 531

Appendices & Index

APPENDIX ONE - Zorlib **535**
The Zortech Librarian 535
What is a librarian 535
Using Zorlib 536
Using Zorlib with Interactive Prompts 538
Using Response Files 538
Capacity 539
Notes 539

APPENDIX TWO - Error Messages **541**
Compiler Error Messages 541
ZTC1, ZTCPP1 and ZTC1B 541
C++ Extension Errors 542
Standard C Errors 545

Link Error Messages 556
Technical Information 559
Zorlib Error Messages 561
System Errors 562
Make Error Messages 563
Global Optimizer Error Messages 565
Runtime Error Messages 566

APPENDIX THREE **567**
Converting from Previous Versions of the Zortech C compiler . 567
Common Blocks 567
Chars 567
Memory Models 567
Predefined Macros 568
Preprocessor 568
Far Pointer Arithmetic 568
Changed Library Functions 569
Linking 569
Assembly Language Interface 570

APPENDIX FOUR **571**
Complete ASCII Chart 571

INDEX **575**

PART ONE

the user guide

QUICK START

Quick Installation

Before you can begin to write C++ programs using Zortech C++, it will be necessary to set up your disks correctly. This quick installation guide is designed for hard disk users. For more information on installation refer to the *Detailed Installation Guide* in Part Two. The file ZTCSETUP.EXE on the compiler disk will install Zortech C++ on your hard disk. It will create the directory C:\ZORTECH and a number of sub-directories.

Before you run ZTCSETUP you should make copies of the distribution disks and use these working copies for the installation. You should then put the distribution disks and working copies in a safe place. To run ZTCSETUP place the working copy of system disk #1 in drive A:. If you are not at the A> prompt type A: and press Enter. At the A> prompt type:

```
ZTCSETUP
```

and press Enter. Follow the displayed instructions. As the files from each disk are copied, you will be prompted for the next disk. You should ensure that you insert the correct disk at each point.

After all of the disks have been copied, alter your AUTOEXEC.BAT file to include the following

```
SET PATH=C:\ZORTECH\BIN
```

Or add C:\ZORTECH\BIN to the current path.

```
SET INCLUDE=C:\ZORTECH\INCLUDE
```

Directory containing include files.

```
SET LIB=C:\ZORTECH\LIB
```

Directory containing library files.

If you have a RAM disk you can also add:

```
SET TMP=d:
```

Where d: is the drive number for your RAM disk. It is also recommended that the following two lines be added to your CONFIG.SYS file:

```
BUFFERS=20
FILES=20
```

If you want the Context Sensitive Help to be loaded automatically you should also put the following line in your AUTOEXEC.BAT file.

```
ZTCHELP
```

Now reboot your computer to install the new environment.

Quick Editing

Starting up in the program editor is simple, provided your system has been set up as described earlier. Change to the directory in which you wish to keep your source files and type:

```
ZED
```

You will then find yourself in the program editor. If you wish to specify a file to edit at startup, you can use the following alternative syntax (assuming your file is called MYPROG.CPP):

```
ZED MYPROG.CPP
```

In this case, if the file does not exist it will be created, and opened for editing, otherwise the existing file is opened. You may now start to type in your C++ program.

Note that C++ files normally have the extension .cpp. ZED will automatically use the C++ compiler for files with this extension, and the C compiler for files with a .c extension. The use of the C++ compiler for all files can be forced by the use of the -cpp compiler flag (see the reference guide for details). C++ header files are normally given the extension .hpp.

Start up ZED as described above and type in the following program :

```
#include <stream.hpp>
main()
{
    cout << "Hello Bjarne!\n";
}
```

check that you have typed it in correctly and then select the menu by pressing **Esc**. Use the arrow keys to move the highlight bar to the menu option **Others** and press **Enter** (or simply press the letter O). Now in the same way select the option **cOmpile** from the next menu.

ZED is now asking for any compiler flags or other information, such as additional library files to use. Pressing **F1** at this point will give you a list of the available options, and what they mean. If there is any text displayed in the input area already, it can be removed by pressing **F9**. ZED remembers the compiler flags used at the last compilation and puts them in as default. Type the compiler flag for the compact memory model **-mc** and press **Enter**.

Assuming that you have typed the example correctly, your program will now compile. If an error is encountered during compilation, control will return to the editor with the cursor placed on the line where the error occurred. You should note that the cause of the error may be on an earlier line, but not detected by the parser until later in the compilation.

When compilation is complete you will be given the option of entering any command line arguments that your program requires, pressing **Enter** to run the program immediately or pressing **Esc** to return to editing. If you have not saved your program you should normally escape at this point and save it using the **FS** option. In the case of our example which is unlikely to cause any problems, you may just press **Enter** to run the program. You should see your message to Bjarne displayed, together with a prompt telling you to press any key to return to ZED. You have now compiled and executed your first program using Zortech C++.

Exiting the program editor is also easy. Simply press escape to go into menu mode (if you are not already in it), and type **FX** (for File, and eXit). A prompt will be given to confirm saving of any altered buffers. For more information on compiling, and linking C++ programs, refer to *Part Two - The Reference Guide.*

INTRODUCTION TO C++

Throughout this manual, we will assume you have a working knowledge of the C language. If you don't, there are dozens of books available on the subject. *C Primer Plus* by Waite, Prata, and Martin is a good choice, and has a chapter on C++.

If you already know C, you'll find C++ easy to pick up because you don't need to learn it all at once. You can program just as you would in C and add C++ features as you learn them.

There are two other books you should own. *An Introduction to Object-Oriented Programming and C++* by Richard Wiener and Lewis Pinson, Addison-Wesley 1988, introduces the C programmer to C++ and includes several useful examples. *The C++ Programming Language* by Bjarne Stroustrup (the creator of C++), Addison- Wesley 1986, is the "expert's guide" to C++ and contains many small but subtle and advanced examples. It's where you look for the tricky stuff, and it's the "last word" on language questions. Both books are available from Zortech.

The next two chapters are intended to give you a feel for the C++ programming language, and Zortech C++ in particular, and to introduce the additional features that C++ has to offer over its predecessor C. They are not intended as an in depth tutorial, and we recommend that you buy one or both of the books mentioned above, in order to gain proficiency in the language. These books are both available from Zortech.

Object-Oriented Programming

The goal of object-oriented programming is to make you more productive. The language helps you organize your code, and it helps you re-use code that you or someone else has written. The less work you have to do to write a program, the more productive you are. Object-oriented languages let you create something called a *class*. A class defines a new type of data. When you've finished defining a class, the compiler treats it just as if it were a type of data which was built into the language, like an integer or a floating-point number. The compiler does all the things for your new class that it would do for a built-in type, including error checking. Defining new classes is often called *abstract data typing*.

If you have a class, and you need one which is slightly different, you don't need to go in and change the class. You can simply say: "define a new class which is this old class over here plus some changes and additions I want to make." This is called *inheritance* or *type derivation* (since you are "deriving" a new type from an old one, or "inheriting" features from an old type). Defining classes this way saves a lot of work.

An object-oriented language needs more than abstract data typing and type derivation. If you start with a single class (often called a *base* class) and derive a number of classes from it, you must be able to control the derived classes through the base class. For instance, if you have a base class called **`shape`** and you derive classes called **`line`**, **`circle`** and **`square`**, each derived class has a different method of drawing itself. You can make a list of **`shapes`**, however, and **`draw`** each one without knowing whether it's a circle, a square, etc. This ability is often called "commonality" or "polymorphism." It means: the object decides, at run-time, what to do with the message you've sent it (commonality isn't automatic - see the section on virtual functions). Thus, a brief definition of object-oriented programming might be:

Object-Oriented	=	**Abstract Data Typing**
	+	**Type Derivation**
	+	**Commonality**

You'll understand the meaning of these terms as you learn the features of the language. For now, just remember that C++ will help you program more productively.

Efficiency

In the past, object-oriented programming languages have been very inefficient. While the programmer can create programs more quickly, those programs have often run slowly and required a lot of computer resources. Also, you generally needed the entire development environment loaded to run the program. This made "serious" program development unwieldy.

When Bjarne Stroustrup at Bell Laboratories set out to create C++, he refused to sacrifice the speed and memory efficiency of C. He also adopted C's "minimalist" approach: there are just enough features in the core of the language to do everything you can imagine. If that isn't the way you want it, you can "extend" the language yourself by creating new classes.

C++ thus gives you the run-time efficiency of a compiler combined with the development advantages of an object-oriented programming system.

Quick Pre-requisites

Declarations vs. Definitions

This is actually C, but it's important to understand the distinction between declarations and definitions, especially when talking about C++. A *declaration* is a message to the compiler. It generates no object code. It simply says: "here is what this function looks like: its name, what it returns, and the proper way to call it. You'll see the actual code sometime later." A declaration tells the compiler that it *isn't* an error when you see this function called, and it *is* an error if the function isn't called in just this way. Here is a declaration:

```
double foo(int i, long j, float bar);
```

In plain C, if you didn't declare a function before using it, that was OK: the compiler assumed it returned an `int`, and it didn't check the parameters you passed it so you could pass any number or type. In C++, type checking is strong. There are very few cases where you can get away without declaring a function before using it. A *definition* is the code which says exactly how the function works.

Function Prototyping

In the above declaration, you'll notice that not only the name of the function was given, but also the type of all the parameters. This will look familiar to Pascal programmers; it's called *function prototyping*. It allows the compiler to determine if you've made a function call correctly. In plain C, you didn't give parameters when declaring a function. When making a call to a function, you could pass it any number or type of parameters, and the compiler wouldn't care -- this caused all sorts of hard-to-find errors. The new ANSI C standard (and the Zortech C compiler, which follows the standard) requires function prototyping.

Zortech C++ performs *Autoprototyping*. If you don't declare a function before using it, the compiler looks at the first call to the function and infers the prototype.

Using Header Files to Declare Classes

When creating a new data type, you declare an entire class: the name of the new data type, what data it contains and what kind of functions work with it. These class declarations generally go in a header file. Anyone who wants to use the class simply includes the header file. This tells the compiler that the class exists and what you can do with it. If you try to do something incorrectly, the compiler will tell you.

As an example, suppose you have a class called a **box** which you can **fill** with three integers, and then **empty**. Zortech C++ header files usually end with the suffix **.hpp**, so let's assume the **box** header file is called **box.hpp**. Here is an example of how you might use boxes:

```
#include "box.hpp"
                    /* tells the compiler what a box is */
main() {
    box B1, B2;                        /* make some boxes */
    B1.fill(1,2,3);
    B2.fill(10,11,12);                  / * fill them up */
    B1.empty(); B2.empty();             /* dump them out */
}
```

As you can see, using a class someone else has defined is pretty simple.

Quick Introduction to Streams

"Streams" are a pre-defined set of classes for console and file I/O. They are more intuitive and much easier to use than the library functions in plain C (although you may certainly continue using **printf()**, **scanf()** and the like if you wish -- C++ doesn't prevent you from using plain C functions). In plain C, standard I/O is performed with *standard input* (**stdin**), *standard output* (**stdout**) and *standard error* (**stderr**). With streams,we have the equivalents in **cin**, **cout** and **cerr** (plus **cprn** and **caux** for the printer and auxilliary ports).

The basic idea is this: here are some objects which do I/O for you. All you have to do is send stuff to them, and they take it from there. To hand things to stream objects, we use the"<<" and ">>" operators, which have been given a special meaning when used with streams (this is called *operator overloading*; see the section on *Operator Overloading*). As an example, let's send some numbers to standard output:

```
#include <stream.hpp>
                    /* declare the streams library */
main() {
   int a = 1, b = 2, c = 20;
   double pi = 3.14;
   cout << a; /* send out the value of a. You can */
   /* include literals, and string things together */
   cout << "b = " << b << "\n";
   cout << "c should be " << 20 << "c = "
            << c << "\n";
     /* What happens when you run things together? */
   cout << a << b << c << pi << "\n";
}
```

You can also use streams for input, for file manipulation and to move things around in memory. You can write stream functions for your own objects. We've just shown you the basics here because we use streams a lot to print values in our examples. There are stream examples in the next chapter, and a more thorough coverage in *Stroustrup* and *Wiener and Pinson*.

'lasses - Defining Your Own Types

Now let's take a look at what makes C++ so different from C: the ability to define your own data types.

Defining data types ("*abstract data typing*") is a powerful concept because it allows you to hide the complexity of your program. This makes the program easier to understand and debug. You can also hide complicated data structures inside classes, and make sure no one is able to change that data but you -- if data gets changed, *YOU* must have changed it. This helps isolate bugs.

Access to items in a class is controlled with the `public` keywords. If something is `public`, everyone has access to it. *private* items are only accessible to you, the creator of the class.

One more piece of terminology is important before we create our first class: the difference between the C++ keyword `class` and the word *object*. A `class` defines a new data type: what it looks like and what functions can be performed upon it. An *object* is one *instance* of that class; you can have many others. A class is a toaster factory; toasters are the objects that come from the factory. So an example can be as follows:

```
class ToasterFactory {
   int number_of_slices;
   float brown_ness;
public:  /* above data is inaccessible to the user */
                  /* the user can call these function
                                  to change the data */
   void insert_toast(int slices);
   void set_brown_ness(float brown_level);
   int is_ready();
   void pop_up();
};   /* Now use the class to create some objects. The
         code for the functions appears elsewhere. */
main() {
   ToasterFactory Toaster1, Toaster2, Toaster3;
   Toaster1.insert_toast(2);   /* put in two slices */
   Toaster2.set_brown_ness(4.2);
   if (Toaster3.is_ready()) Toaster3.pop_up();
}
```

In the example, we've described what a toaster can do, and we've told **ToasterFactory** to create three toasters. Then we told each toaster to do something. Notice that the data elements appear before the **public** keyword. This means they are **private**; in C++ everything reverts to **private** unless you explicitly call it **public**.

Notice also that the functions are tied to the class. You cannot just say **insert_toast(2)** without associating it with an object. For this reason, functions which are declared inside a **class** are called *member functions*.

Before our second example, a note of explanation. We will often show you class declarations and code which will appear to be in the same file. Stroustrup also does this in *The C++ Programming Language*, and it can be confusing. They are presented this way for simplicity, and you can indeed type them in and run them exactly as they are shown. However, when *you* write code, the class declarations should all go in **.hpp** files, and the class definitions (i.e.the code) should go in **.cpp** files (which include the **.hpp** files). This makes classes much easier to use in other programs.

A Complete Example

Now let's create a class called `OriginalWork`,which represents a piece of creative work. We'll say it has an author, a title, a date of creation and a value. We'll call the file `OWORK.HPP`.

```
#include <stream.hpp>
class OriginalWork {
   char author[20];
   char title[20];
   int month, day, year;
   int value;
public: /* first, something called a "constructor" */
   OriginalWork ( char * Auth, char * name, int m,
                  int d, int y, int val);
void print();
};   /* semicolon after closing brace is essential!*/
```

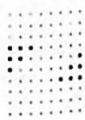
Here we've slipped in something new: a *constructor*. Notice the constructor has the same name as the class.

The compiler calls the constructor when an object is declared, so you can put code in the constructor to perform automatic initialization. In our constructor, it appears that all values in the object will be initialized. Let's write code for `Original Work` in the file `OWORK.CPP`. It's just like writing C code except;

- Each *member* function name must be associated with the class it is written for using the C++ operator ::
- The member functions have access to elements in the class data structure.

```
#include "owork.hpp"
            /* quotes for files in this directory */
#include <stream.hpp>
           /* angles for files in the include dir */
#include <string.h>
OriginalWork::OriginalWork (char * Auth, char * name,
                        int m, int d, int y, int val)
{
   month = m; day = d; year = y;
   value = val;
   strcpy(author, Auth);
                    /* calling a plain C function */
   strcpy(title, name);
}
void OriginalWork::print()
{
   cout << "name: " << title << "\n";
   cout << "Author: " << author << "\n";
   cout << "Created on: " << month << "/" << day << "/"
                              << year << "\n";
   cout << "Value : $" << value << "\n";
}
```

The constructor is unusual in that you don't have to say what its return value is -- the compiler already knows (a pointer to the object itself).



Finally, we can use the class (call this file **MAIN.CPP**):

```
#include "owork.hpp"
main()
{
OriginalWork painting("Bob Rembrandt",
                          "Blue Boy",5,24,65,74);
OriginalWork photo("Eisenstadt", "Moon",
                          6,30,77,2000);
OriginalWork chapter("Bruce Eckel", "Intro to C++",
                          5,24,88,0);
painting.print();
photo.print();
chapter.print();
}
```

Since we showed this program in three separate pieces (**OWORK.HPP**, **OWORK.CPP** and **MAIN.CPP**, you will need to compile **OWORK.CPP** and **MAIN.CPP** into *object modules*, and link them into a finished **.EXE** file, called ART.EXE using the command:

ZTC -oART.EXE MAIN OWORK

This will also generate the object files **OWORK.OBJ** and **MAIN.OBJ**. You can also just jam everything (header files and all) into a single file called **ART.CPP** and run the command:

ZTC ART

This can be very convenient for small programs and for testing things out. However, you'll find that most projects will consist of a lot of classes, all kept in different header and code files, many of which are already compiled into object modules. Most of the work of bringing a program together will simply be including the header files, using the classes inside your **main** routine, and linking in all the object modules containing the compiled code for the classes.

Notice we are using the term "object" here in the historical sense, not the object-oriented sense: the *source code* is the "starting point," and the compiler's "objective" is to create *object modules* of compiled code, with the extension `.OBJ`. This terminology is confusing and unfortunate, but it was developed before anyone thought of object-oriented programming.

To create a program which consists of many independently- compiled object modules, you could write an MS-DOS *batch* file, with all the necessary commands. This would save you typing, but not time: every time you created your program, all the source code would be re-compiled, whether it was changed or not. There is a smarter way to do things: see the chapter titled "*TOOLS*" for a description of the `make` utility.

Function Overloading

In our second complete example, we'll introduce the concept of *function overloading*. This means you can have many functions with the same name but different arguments, and the compiler will choose the correct function. This is a very nice feature, because it hides details and makes your code more readable.

Probably the most common example of function overloading is constructors. There are usually a number of ways to make an object. For instance, let's create a class called `date`; it can be initialized with 3 integers, a string representing the date, or no value at all (in which case it uses the current date).

```
#include <stream.hpp>
class date {
   int month, day, year;
public:
   date();     /* no initial value --> today's date */
   date(int m, int d = 1, int y = 88);
   date(char * datestring);
   date(date &);
   void print(char * msg = "");
};
```

You'll see something else that's new: in the second constructor, **d** and **y** have *default values*, which means if you say **date birthday(12)**, **birthday** will automatically be given values of **d = 1** and **y = 88**. You can also say **date birthday(11,5)** or **date birthday(12,5,87)**. Default values must always be the *last* values in the list. The fourth constructor is called a *copy-initializer*. It's special, and it often pops up where you don't expect it. For instance, if you make a declaration **date celebration=birthday;**, the copy-initializer is automatically called, even though the statement doesn't *look* like a call to that constructor (which would appear **date celebration(birthday);**).

The lack of a copy-initializer is often the cause of problems with beginners; it's so important that the next AT&T C++ definition will automatically generate a copy initializer if you don't write one yourself. Now let's look at the code for the **date** class:

```
#include "date.hpp"
#include <time.h>
#include <stdlib.h>                    /* for atoi() */
date::date() {
   struct tm *t;
   time_t ltime;
   time(&ltime);
   t = localtime(&ltime);
   month = t->tm_mon+1;
   day = t->tm_mday;
   year = t->tm_year;
}
date::date(int m, int d, int y) {
                    /* notice we only give the default
                              values in the class decl */
month = m; day = d; year = y;
}
date::date(char* datestring) {
   char buf[10];
                /* temporary holder for the pieces */
   char * dt = datestring; char * bf = buf;
```

(Continued Overleaf . . .)

(Continued)

```
    month = atoi(dt);
                    /* converts until unrecognized char */
    while(*dt != '/' && *dt != ':') dt++;
    dt++;                           /* move past '/' or ':' */
    day = atoi(dt);
    while(*dt != '/' && *dt != ':') dt++;
    dt++;
    year = atoi(dt);
}

date::date(date & otherdate)
{
                                    /* the copy-initializer */
month = otherdate.month;
day = otherdate.day;
year = otherdate.year;
}

void date::print(char * msg) {
    cout << msg << " date: " <<
        month << "/" < day << "/" < year << "\n";
}
```

The copy-initializer appears to do something surprising - it treats its argument as if it were a structure, and reaches in to grab elements of the structure directly. To C++, an class *IS* a structure with special characteristics, but still a structure.

The only way to directly access elements of any class (even one passed to us as a parameter) is if we've been explicitly given permission - we must either be a member function, or a **friend** (see the section on *Friends*) of the class we are manipulating.

Now we have several kinds of constructors we can call:

```
#include "date.hpp"
main() {
   date today;                          /* calls date(); */
   date also_today(today);   /* calls date(date &); */
   date today_too = today;   /* calls date(date &); */
   date long_ago(1,1,450);
                        /* calls date(int, int, int) */
   date far_ahead("7/8/2025");
                           /* calls date(char *); */
   today.print("today");
   also_today.print("also_today");
   today_too.print("today_too");
   long_ago.print("long_ago");
   far_ahead.print("far_ahead");
}
```

Operator Overloading

Since built-in types like `int`s and `float`s can be acted upon by operators, the types we create as programmers must also be able to be used with operators. If you want an operator to work with your new type, you use *operator overloading*. Overloading means "it has more than one meaning, depending upon the context." For instance, the '<<' operator is overloaded in the `streams` class. We'll make a silly example called a `tuit` so you can see how to overload operators:

```
#include <stream.hpp>
class tuit {
   int a, b, c;
public:
   tuit(int i=0, int j = 0, int k = 0)
   { a = i; b = j; c = k; }
                        /* see "in-line functions" */
   tuit(tuit &);                /* copy-initializer */
   tuit operator+(tuit &);
   tuit operator-(tuit &);
   tuit operator=(tuit &);
   void print(char * name = "");
};

tuit::tuit( tuit & othertuit) {
   a = othertuit.a; b = othertuit.b; c = othertuit.c;
}

tuit tuit::operator+(tuit & arg) {
   tuit result;
   result.a = a + arg.a;
   result.b = b + arg.b;
   result.c = c + arg.c;
   return result;
}
```

(Continued Overleaf . . .)

(Coninued)

```
tuit tuit::operator-(tuit & arg) {
   tuit result;
   result.a = a - arg.a;
   result.b = b - arg.b;
   result.c = c - arg.c;
   return result;
}

tuit tuit::operator=(tuit & rvalue) {
   a = rvalue.a;
   b = rvalue.b;
   c = rvalue.c;
   return *this;
}

void tuit::print(char * name) {
   cout < name < ": ";
   cout < "a = " < a < ", b = " < b <
      ", c = " < c < "\n";
}

main() {
   tuit x(1,2,3), y(10), z(11,12);
   tuit u = x + y - z;          /* Pretty slick, eh? */
x.print("x"); y.print("y"); z.print("z");
(x+y).print("x+y");
(x+y-z).print("x+y-z");
u.print("u");
}
```

There are two concepts in this example which you may find puzzling: the use of the '&' and the keyword `this`. The meaning of '&' is essentially the same as in plain C: "address of." You can use it as you would in C (if you have an `int c` you can get the address by saying `&c`). In C++, however, the use of '&' is extended -- in the example it's called a *reference*. Values are being passed in and out of the functions "by reference." It means: "I'm going to hand you a value, turn it into an address." Pass-by-reference is the preferred method of passing parameters and returning values; it's generally more efficient.

The keyword `this` means "the address of the object I'm in." Member functions can easily access any element of the object simply by naming the element, but `this` is necessary to get the object's address. When overloading operators, it's always important to return a value. The value is fairly obvious in the case of `operator+()` and `operator-()` (the result of the operation) but not so apparent with something like `operator++()` or `operator=()`.

To solve this little dilemma, remember you must be able to use the operators in expressions like `if(A++)` and `A=B=C`. Thus, the result of applying the operator to the object should be the object itself: `*this`. Notice the addressed must be de-referenced first before returning it; the function was declared to return a reference and a reference expects to be given a *thing*, not the address of a thing.

Making a New Type From an Old Type

You might be thinking at this point that there's a fair amount of concentration involved in creating a new type. There is, by the time you add operators and all the other slick features. But this concentration pays off in two places:

1. Once you get a new type working, it *always* works. It's easy to use in your final system because the compiler has ensured that the interface is correct and that you use it properly.

2. Once you have a working type, you can *inherit* it into new types. You can make small or large modifications to your new type, but most of the work is already done.

Let's see how this works by inheriting from the class `OriginalWork`. We'll make a class which is a more specific creative work:

```
#include "owork.hpp"      /* declare the base class */
class Painting : public OriginalWork {
                                  /* inherit the base */
   char style[30];            /* add some new members */
   char owner[30];
public:
   Painting(char * artist, char * title, char * sty,
            char * own, int m, int d, int y, int val);
   void information();
                        /* tell us about the painting */
};
```

The only difference between `Painting` and an ordinary class definition is the "`: public OriginalWork`," which says: "this new data type is made from `OriginalWork`." `public` allows the derived class `Painting` to have access to all `public` elements of the *base class* `OriginalWork`. We now define the member functions:

```
#include "painting.hpp"
                        /* where the declaration lives */
Painting::Painting( char * artist, char * title,
                        char * style, char * owner, int m,
                        int d, int y, int val)
        :  (artist, title, m, d, y, val) {
           strcpy(style,sty);
           strcpy(owner,own);
}
void Painting::information() {
    cout << "Painting:\n"
    print();
    cout << "Style : " << style << "\n";
    cout << "Owner : " << owner << "\n";
}
```

Two tricky things have happened here, the constructor for the base class has been called, and a base class member function was called. The constructor for the base class is called after the function name and parameter list and before the opening brace of the function. It is delimited by a ':', just as the inherited class is delimited in the class declaration. If your base class has an overloaded constructor, you can call any one of the constructors simply by specifying the appropriate parameters. In `Painting::information()`, we called `print()` which is a member function of `class-OriginalWork`. It's that simple - if a function exists in the base class, you can use it. Here's something to exercise our new class:

```
#include "painting.hpp"
main() {
    Painting Bull("Ralph Picasso", "Bull & Doves",
          "Abstract Expressionism", "Bend Art Museum",
          5,26,88, 495);
    Bull.information();    /* call a member function */
    Bull.print();
            /* call a function from the base class */
}
```

To show you this sort of thing can go on without end, let's derive a class from **Painting**:

```
                                        /* File CUBIST.HPP */
#include "painting.hpp"
class Cubist : public Painting {
   int number_of_eyes; /*... on one side of the face*/
public:
   Cubist( char * artist, char * title, char * sty,
           int eyes, char * own, int m, int d,
           int y, int val);
   int eyes() { return number_of_eyes; }
};
Cubist:Cubist( char * artist, char * title,
               char * sty, int eyes, char * own, int m,
               int d, int y, int val)
   : (artist, title, sty, own, m, d, y, val) {
      number_of_eyes = eyes;
}
#include "cubist.hpp"
main() {
   Cubist Only("Bob Dega", "I Only Have Eyes...",
               "early", 3, "Bend Art Museum",
               1,14,88, 196);
   Only.information();
   cout  << "This was from the " << Only.eyes()
         << " eye period\n";
}
```

You may only call constructors from the immediate base class, not the class the base class was inherited from.

eyes() was written *in-line*. This is a convenient and efficient way to define very short functions (see ***in-line***).

Finally, what if you don't want to inherit a class, but have an object of that class as a member of your class structure? E.g. if we want to use our **date** class to record the start and end of an artist's particular period. Inheritance doesn't work here, since we want to use **date** twice. The answer is to use *member objects*:

```
#include "cubist.hpp"          /* the above example */
#include "date.hpp"     /* declare the "date" class */
class CubistPeriod : Cubist {
   date beginning;
   date end;
public:  /* Note how the base class and member object
     constructors are called before the left brace */
   CubistPeriod(char * artist, char * title, char * sty,
         int eyes, char * own, int m, int d, int y,
         int val, char * start, char * finish)
      :  (artist, title, sty, eyes, own, m,d,y, val),
         beginning(start), end (finish){}
           /* This is probably bigger than an in-line
                                  function should be: */
   void All_info() { information();
      cout << "period starts: "; beginning.print();
      cout << "period ends: "; end.print();
   }
};
main() {
   CubistPeriod Dog("Bob Dega", "Five-legged dog",
         "middle", 2, "Bend Art Museum", 2,24,88, 399,
         "11/15/87", "2/25/88");
   Dog.All_info();
}
```

The compiler will tolerate a fair amount of abuse with in- line functions, but it gives up if you make them too big or too complicated. As you can see, calling the constructor for a member object is almost the same as calling the constructor for the base class, except you *name* the object.

'ou've Been Through Basic Training

At this point you've humped it through the C++ beginning obstacle course. You have enough exposure to begin writing your own classes. As you see how easy it is to use C++, you will start getting new and more daring ideas, and you'll want to know how to do things we haven't shown you yet. The next chapter covers more sophisticated features of C++.

lore Help is on the Way...

As an owner of Zortech C++, you will begin receiving a one-year subscription to the *Zortech C++ Newsletter* as soon as you send in your registration. The newsletter contains programming tips, tricks and traps when using Zortech C++, as well as information about new releases of the compiler, C++ products, and other useful C++ news.

If you don't own the compiler or your subscription has expired, you can receive the newsletter by sending $32 or £15.75 for a one- year subscription of six bi-monthly issues to:

AMERICA	EUROPE
Zortech C++ Newsletter	**Zortech C++ Newsletter**
361 Masachussetts Ave	**106-108 Powis St**
Arlington MA 02174	**London SE18 6LU**

If you submit an article which is printed in the next newsletter, you will receive a one-year subscription and your next update of the Zortech C++ compiler free.

SOPHISTICATED C++

This chapter introduces some of the more advanced features of the language. The treatment of the subjects is not meant as a complete reference; only enough to familiarize you. For more details, please see the books by *Wiener and Pinson* and *Stroustrup*.

'alling Member Functions

As we've seen before, if you have an object `OBJ` and you want to call one of its member functions `func()`, you simply say `OBJ.func()`. What if you're defining a member function for a class, and you want to call another member function? When you're inside a member function, it's always assumed that you're working on an object, but it doesn't have a name yet - naming doesn't occur until the object is declared. To call a member function inside another member function, simply call the function with no object associated with it, e.g.: `func()`.

'onstants

To create a constant value in plain C, you used the preprocessor, for example: `#define A 10`. Unfortunately, once the pre-processor is finished substituting "10" for every `A` it sees, the compiler has no record of `A` and cannot give you any error messages about `A`, which makes debugging much harder.

C++ has the concept of a `const`, which tells the compiler to prevent any modification of a value. In the above case: `const int=10;`. Constants are much more flexible than the `#define` statement; they understand "scoping" (see section on scoping), you can declare an object to be constant, and you can pass constant parameters to functions. Zortech C++ can generate better code it knows something is a constant.

Comments

C++ has two kinds of comments: the usual C /* and */ and a new type of comment which only goes until the end of the line:

```
int a = 4;          // only goes to the end of the line
```

Multiple Inclusion of Header Files

Whenever you use a class, the header file which declares the class must be included. Since some classes inherit or use other classes, header files from one class may include header files from another class, and you may end up including the same header file more than once without knowing it. The compiler will not accept a re-declaration of the same class and will flag it as an error.

A common practice used to prevent this error is to use the pre-processor to avoid declaring a class more than once. This is accomplished by creating a special pre-processor variable for each included file. The file name, in all capital letters, is used with the '.' removed (ANSI C doesn't allow the use of the '.' in variable names). Here's an example for a file called "CHAIR.HPP":

```
#ifndef CHAIRHPP
#define CHAIRHPP
class chair {           // declaration for class chair
};
#endif CHAIRHPP
```

The declaration for **chair** is only included if the variable **CHAIRHPP** has not been previously defined.

coping

Scoping rules describe the parts of a program where a variable is valid. These parts are delimited by opening and closing braces (`{` and `}`).

To see when a variable *goes out of scope* (becomes invalid), check for the nearest opening brace *before* the variable is declared, then find the closing brace which matches it. The variable goes out of scope when the closing brace is encountered. If the variable is an object, it's destructor will be called (if one has been defined) when it goes out of scope.

In plain C, all variables must be declared at the beginning of the scope, right after the opening brace and before any code is written. In C++, you may declare variables at the point where you use them. A convenient example is the counter in a `for` loop:

```
...
for(int i = 0; i < 100; i++)
   for(int j = 0; j < 100; j++)
   // the work goes here
```

Notice that you can declare and initialize a variable at the same time.

Inline Functions

There are many situations where you would like to hide some programming complexity by encapsulating a small piece of code in a function call (this also allows you to use a name which might make more sense to someone reading the code). Unfortunately, function calls have a certain overhead which is often undesirable.

C++ allows the best of both worlds via `inline` functions. By declaring a function `inline`, you tell the compiler to replace all function calls with the code in the inline function (just as if you had used the preprocessor).

Inline functions are generally considered the only "acceptable" definitions to appear in header files. They should be kept short. Inline functions in C++ should be used in places where plain C would have used `#define` to replace a short section of code. Examples appear in the next section.

Access Functions

Normally, data in an object is *private*, which means you can't get in and muck around with it unless the programmer specifically gives you permission (i.e. you must use a member or friend function). Often, however, you want to read certain values in an object. Instead of making these values public (which means *anyone* can change them), it's better to write *access functions*, which return a value without giving the outside world permission to change it. Access functions are probably the most common use of *inline* code. Here's an example:

```
#include <stream.h>            // so we can use streams
class insect {
   int legs;
   int eyes;
   int flight;
public:                 // the stuff above is all private
                              // an inline constructor:
   insect(int l, int e, int f)
                              {legs=l; eyes=e; flight=f;}
                        // some inline access functions:
   int number_of_legs() {return legs; }
   int number_of_eyes() {return eyes; }
   int does_it_fly() {return flight;}
};
main() {
   insect bug(7,9,1);
   cout << bug.does_it_fly();
}
```

Note that you don't have to declare a function `inline` if it is part of the class declaration; it is automatically assumed.

Dynamic Memory Allocation

Memory can be made available in two ways in C++. When variables are declared, they are created on the *stack* by pushing the stack pointer down. When the variables *go out of scope*, the space for the variables is freed automatically by moving the stack pointer up. The size of stack-allocated memory must always be known at compilation. If you need to make a variable with a size unknown at compilation, you must allocate the memory for that variable yourself, on the *heap*, also called the *free store*. The heap can be thought of as occupying the bottom of the program's memory space and growing upward, while the stack occupies the top and grows downward.

You may allocate and release heap memory at any point. It is important to realize that heap-allocated memory variables are not subject to scoping rules, as other variables are (this is a useful feature - see section on *References*). These variables *never* go out of scope, so once you allocate memory on the heap, you are responsible for freeing it. If you continue to allocate heap space without freeing it, your program will (mysteriously) crash. Most C packages have library functions (e.g. **malloc()**, **free()**) to provide dynamic memory allocation, but in C++ it was considered important enough to make it part of the core language. C++ uses **new** and **delete** to allocate and free dynamic memory. The argument to **new** is an expression which returns the number of bytes to be allocated; the value returned is a pointer to the beginning of the memory block. The argument to **delete** is the starting address of the memory block to be freed (the same address previously returned by **new**).Let us suppose you have arbitrary integers **x** and **y** and you want to create an **x** by **y** matrix of floating-point numbers:

```
float * mat = new float[x*y];
```

will allocate the matrix, and

```
delete mat;
```

will free the space. You can continue to use the space after freeing it, and do so more than once - both practices are errors and can cause hard-to-detect bugs.

Constructors & Destructors

Any data type may have some form of initialization performed on it when declared, and some form of clean-up when it goes out of scope. When new data types are created in C++, this initialization and clean-up is performed via *constructors* and *destructors*.

The compiler automatically calls the correct constructor when the object is declared, and the destructor (if there is one) when the object goes out of scope.

Examples of initialization include setting variables to known values, allocating heap memory, copying values from another object, and more unusual tasks associated with the esoteric types you can create with C++ (i.e. drawing a window, clearing a queue, initializing a serial port). Examples of clean-up include de-allocat ing heap space, removing a window from the screen, etc.

While there can be many types of constructors (different according to parameter lists), there is only one destructor for each class. Destructors have no parameter lists. Constructors use the same name as the class. Destructors use the class name with a tilde prepended. Here are some example constructor and destructor declarations:

```
class matrix {                          // matrix data
public:
   matrix();                         /* No arguments */
   matrix(matrix &);              /* copy-initializer */
   matrix(int i, int j, int a);
   matrix(int i, int j , float a);
   ~matrix();                        /* destructor */
                              // other matrix declarations
};
```

Constructors and destructors are not always necessary for a simple class. If your class is based on the stack (no part of is created on the heap), doesn't require variables to be initialized and is only passed to functions using references (passing by value requires a copy-initializer), then a constructor probably isn't necessary. If no clean-up is needed when an object goes out of scope, then a destructor usually isn't necessary.

The code for constructors and destructors is identical to any other member functions, with a few exceptions. The constructor automatically returns a pointer to `this` (the object itself, allowing initialization and assignment at the same time) and the destructor doesn't return anything. Neither return value is declared in the class declaration or when the functions are defined.

Normally, when an object is created, stack space is allocated for any variables which live on the stack and then the constructor is called (conversely, when an object goes out of scope the destructor - if any has been defined - is called, then the stack space is de-allocated). If, however, you make an assignment to `this` in the constructor, automatic stack allocation is overridden (you should know what you're doing before you try this trick).

References

A reference is a pointer which looks like an object. The compiler takes care of de-referencing (applying the * operator to) the pointer for you. You can initialize a reference from another object (in which case you have two ways to manipulate the same object - this introduces problems of its own and should be used carefully). Here's an example:

```
foo A();                                    // create a "foo"
foo & B = A;
          // create a "foo" reference; initialize to A
A.function();        // This does the same thing as this
B.function();
```

In the above code fragment, when **A** goes out of scope, the destructor is called for **A**. Since **B** is a reference, the destructor is never called for **B**.

The fact that scoping is disabled for references can be useful. The second way to use references is by creating an object entirely on the free store, and controlling it through a reference. Most objects live on the stack except for pieces which you explicitly create on the free store via **new**. It is possible to create these objects so they live instead on the free store.

A good example is data objects stored in a binary tree. Generally, you don't want pieces of the tree to be destroyed simply because they go out of scope - you want complete control over when they are added and deleted from the tree.

As an illustration, let's look at how the functions **add()** and **remove()** might appear in a hypothetical binary tree class **BinaryTree**:

```
void BinaryTree::add(int data) {
 // "leaf" is some class which holds data and can be
                    // "attached" to this binary tree
leaf & newleaf = *new leaf(data);
            // Since a reference looks like a thing,
            // and not a pointer to a thing, we must
             // de-reference the pointer returned by
                              // "new" by saying "*new"
attach(newleaf);
}
void BinaryTree::remove(leaf & lf) {
// "un_attach" removes a leaf from the binary tree and
                  // returns a reference to the leaf:
leaf & oldleaf = un_attach(lf);
                  // to delete a reference (i.e. call
                 // the destructor for the reference)
                              // cast it to a pointer:
delete (leaf *)(&oldleaf);
}
```

Note that it isn't necessary to use a reference when creating an object on the free store - you could as easily use a pointer, and the code generated by the compiler would be the same.The only difference is the way the code looks; the resulting objects are still immune to scoping rules.

Using references in the above fashion or as duplicate pointers to an object tends to generate tricky bugs and should be avoided unless you really understand why you are doing it that way. The most common use of references is to pass parameters to functions.

If you declare a function to be pass-by-reference, a pointer (disguised as a reference) is always passed to the function regardless of the syntax of the actual function call. For instance:

```
void fred(foo & A) { A.member_function();}     // use A
main() {
foo B;
fred(B); fred(&B);                             // identical calls
}
```

If a function doesn't modify it's parameters, passing by reference is the preferred method, since it is the most efficient. It's also the simplest way to proceed if you are a novice. If for some reason you *must* modify function parameters (instead of just copying them to a local variable, and modifying that), use pass-by-value.

In this case, the compiler makes a copy of the original (using the copy-initializer, which you must write). This is another method which is subtle and should be avoided by the beginner.

A last trick using references is to disable the information hiding mechanism with an access function to return a reference to the actual value, instead of a copy of a value. An example where this is useful is a vector class which allows the user to change an element of the vector using the `operator[]`.

```
#include <stream.hpp>
class ten_vector{
   int element[10];
public:
   ten_vector(int x = 0) {          /* default value */
      for(int i = 0; i  10; i++)
         element[i] = x;
   }
   int value(int i) { if (i < 10) return element[i];
                 ' else return 0; }
   int & operator[](int i) { return element[i]; }
};

main() {
   ten_vector V;
   cout << V.value(5);
                /* you can read it but not change it */
   V[5] = 20;                      /* use "operator[]" */
   cout << "  " << V.value(5);
}
```

Notice how neatly this works out - since the compiler de-references the address that `operator[]` returns, the line `V[5] = 20` is very readable. Returning an object from a function can be tricky (see the next section). In particular, don't be tempted to return a reference to a local variable (also known as an *automatic variable).* Local variables live on the stack - when the function finishes, you will be returning a reference to a variable which is out of scope!

Operator Overloading

Almost all operators in C++ may be overloaded with meanings for new data types. For the overloaded operators to work properly, certain guidelines should be followed.

- *When passing arguments into operator functions, references should be used and the arguments should never be modified (you wouldn't expect the statement* `A + B` *to modify* `A` *or* `B`*, would you?).*

- *Although it is possible to overload operators as* `friend` *functions with two arguments, the preferred (and less confusing) method is to make them member functions with a single argument. With binary operators,* `this` *is the object on the left of the operator, and the argument is the object on the right of the operator.*

- *When returning an object from an operator function (or any function, for that matter), the return should be performed by value. This means a copy-initializer must be created; in the case of objects with pieces on the heap, this must be done most carefully to avoid generating garbage (an extremely workable method of avoiding garbage is reference counting, demonstrated in the* `String` *class in Stroustrup's book on page 184).*

- *When overloading operators, take care that they work in arbitrarily complex expressions (*`A+B-C/D...`*). The user will undoubtedly try something you don't think of.*

`operator=()`

The function `operator=()` is unusual for a number of reasons. In plain C, assignment returns the value assigned, which allows statements like `A=B=C` *except* when dealing with structures - An idiosyncrasy in the implementation prevents multiple structure assignment. C++ has repaired this problem, so returning the object after assignment (with the statement `return *this`) allows multiple object assignment.

The `operator=()` isn't always used where you expect it to be. E.g:

```
                                        /* file potato.hpp */
class potato {
   int u, v;
public:
   potato (int a, int b = 0) { u = a; v = b; }
   potato (potato & p) { u = p.u; v = p.v; }
   potato operator=(potato & p)
      { u = p.u; v = p.v; return *this; }
};
```

```
                                        // file potatst.hpp
#include "potato.hpp"
main() {
   potato X(1,2);
   potato Y = potato(3,4);          // calls potato(3,4)
   potato Z = 5;                    // calls potato(5,0)
   potato Q = X;                // calls potato(potato &)
   Y = Z;          // calls potato::operator=(potato &)
}
```

Notice the use of `*this` in `operator=()`. Since the return value for `operator=()` is a potato, returned by value, the code `return *this` means "return a copy of myself."

Finally, when deriving a new class, `operator=()` isn't inherited along with all the other functions - it must be re-defined each time a new class is derived.

Often, you want to use the `operator=()` from the base class in the new assignment function without typing it in again. Here's how you do it:

```
#include "potato.hpp"
class tomato : public potato {
   int t;
public:
   tomato(int a, int b, int c) : (a,b) { t = c;}
 // notice the base class copy-initializer is called:
   tomato(tomato & rv) : (rv) { t = rv.t; }
   tomato operator=(tomato & x) {
      t = x.t;
      (potato)(*this) = (potato)x;
      return *this;
   }
};
main() {
   tomato A(1,2,3), B(4,5,6);
   A = B;
}
```

Void Pointers

If you declare a pointer to be `void`, it contains a generic address which can be cast to any type of object. Void pointers are used when passing pointers to functions which cannot know the type of object being passed to them, and for returning untyped objects from functions. Both techniques are subtle, and should be avoided by novices.

Virtual Functions

The *virtual function* is one of the key elements in making C++ an object-oriented language. While a language may support *abstract data typing* and *inheritance*, it can't be considered truly object-oriented unless it supports *commonality* (also called *polymorphism*). Virtual functions are the way C++ supports commonality.

When a function is declared in a base class as an "ordinary" (non-virtual) function, it isn't polymorphic. If you inherit the base class into a new class, and re-define the function, you have created two separate functions. If, however, you declare the function to be `virtual` in the base class and re-define it in classes derived from the base class, there will exist a single function for each class, and the function can be called *for* the derived class, *from* the base class.

What does this mean? Well, suppose you are building a window system and there are three types of objects which can be displayed on the screen (call them `ScreenObjs`): a `Dialog` (which the user may type into), a `Reader` (which the user may scroll through but may not modify) and a `Message` (which displays a message and requests a simple yes-or-no answer). There may be any number of these objects on the screen, all controlled via a screen manager.

The screen manager must be able open, close, move and re-draw all the displayed objects, regardless of what specific type they are. To accomplish this, the functions `Open`, `Close`, `Move` and `ReDraw` are all declared `virtual` in the base class `ScreenObj` and modified in the derived classes.

The screen manager then simply keeps a list of `ScreenObj`s and calls the appropriate function for each item in the list. Here is the way it might look:

```
class ScreenObj{                        // private data here
public:
virtual void Open(){}        // functions MUST be given a
virtual void Close(){}          // definition in the base
                            // class. (even if it's empty)
virtual void ReDraw() {}
virtual void Move(int x, int y) {}   // amount to move
};
class Dialog : public ScreenObj {
// private data for Dialog here
public:
   Dialog();                                // constructor
   ~Dialog();                               // destructor
   void Open(); void Close(); void Redraw();
   void Move(int, int);
};
class Reader : public ScreenObj {
// private data for Reader here
public:
   Reader();
   ~Reader();
   void Open(); void Close(); void Redraw();
   void Move(int, int);
};
class Message: public ScreenObj {
// private data for Message here
public:
   Message();
   ~Message();
   void Open(); void Close(); void Redraw();
   void Move(int, int);
};
```

(Continued Overleaf . . .)

(Continued)

```
// The specific functions are assumed to be defined
// elsewhere (this illustration only shows the frame
// work).They look just like ordinary screen functions

class ScreenList {          // keep a list of ScreenObjs
struct Window {
   ScreenObj * head;            // a private structure
   Window * next; };          // to make a linked list
   Window * head;                        // start of list
   Window * current;    // so we can step through list
public:
   void add(ScreenObj &);             // add to the list
   void remove(ScreenObj &);      // remove a ScreenObj
   void reset() { current = head; } // go to beginning
   ScreenObj & GetNext();
                  // return the next one or zero at end
};
main() {
   Message M; Reader R; Dialog D;
   Screenlist List;
   List.add(M); List.add(R); List.add(D);
   List.reset();
   while ( (ScreenObj & Win = List.GetNext()) != 0 )
      Win.ReDraw();             // ReDraw the whole list
}
```

Destructors may also be declared as `virtual`, to ensure that the proper destructor is called when manipulating derived classes via the base class. Chapter 6 of *Wiener & Pinson* is devoted to virtual functions.

Friends

Situations often arise where you need to give non-member functions access to private elements of a class structure. This access can only be granted by the class itself, by declaring a function (or, in some situations, an entire class) as a **friend** within the class declaration. The following is a simple example:

```
class Integer {
int X;
public
Integer(int I) : X(I) {};
friend void set(Integer &,int);
};
void set(Integer & A, int J) { A.X = J;}
                    // not a member function, but it has
                                        access to private data
```

Notice that in the constructor for **Integer**, the constructor for the member object **X** is called, even though **X** is an int, which is a built-in type.

Public, Private and Protected

The keyword **public** controls access to the data and member functions of a class. **Public** allows access to everyone.

Wiener & Pinson talk about the **private and protected** keywords. These were not part of the definition given in Stroustrup's book and were not supported by Zortech C++ at the time of this manual going to press. However it is intended to support these keywords in future versions of Zortech C++. **Please check the read.me file on distribution disk 2 for more up to date information on new features.**

Static Variables

You can use **static** and *global* objects in Zortech C++. Static objects are subject to normal scoping rules except that the destructor isn't called when the object's function exits, and the object doesn't "go away" between function calls like stack (automatic) variables do.

Global objects are declared outside of all functions and are accessible to all functions. Constructors for static and global objects are called before **main** is entered; destructors are called wherever **exit()** is called, either explicitly or implicitly, following the return from **main()**.

The Scope Resolution Operator

The scope resolution operator **::** allows you to specify exactly which function you want to call in cases where there is more than one choice and the default is not the desired one. When used without a prefix, the scope resolution operator selects the global definition or the one defined by the core language (as opposed to an overloaded definition). As an example, it is possible to overload the heap allocation keyword **new**. To select the "normal" definition instead of the overloaded definition, say:

```
::new
```

Variables may also be selected using the scope resolution operator:

```
int A = 20;
class deelybob {
   int A;
public:
   deelybob() { A = ::A; }
};
```

PART TWO

the reference guide

DETAILED INSTALLATION GUIDE

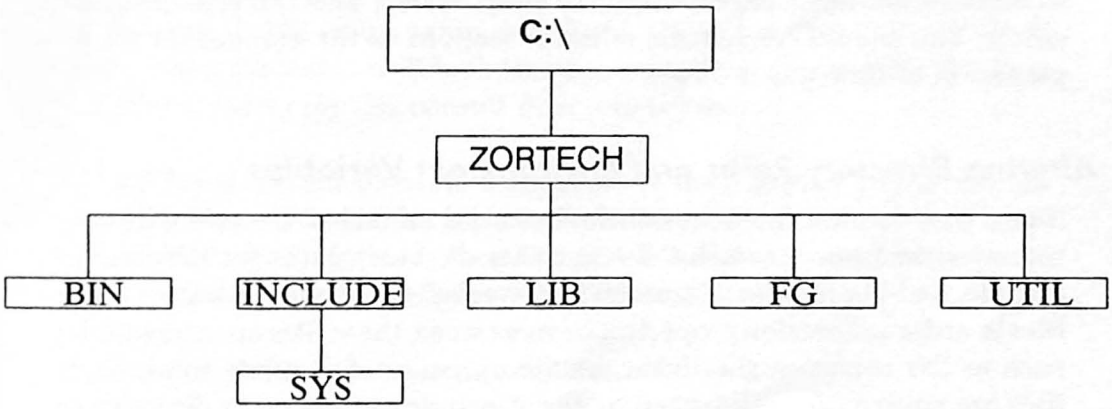
System Requirements

- *A personal computer running MS-DOS 2.0 or later*
- *Two floppy disk drives or one floppy disk drive and a hard disk.*
- *256K RAM available memory.*

Before you can begin to write C++ programs using Zortech C++, it will be necessary to set up your disks correctly.

Hard Disk Installation

The file ZTCSETUP.EXE on the compiler disk will install Zortech C++ on your hard disk. It will create the directory C:\ZORTECH and a number of sub-directories. The directory structure is:



ZTCSETUP then copies the files needed to use Zortech C++ from the distribution disks into the appropriate directories:

C:\ZORTECH	*Main directory contains read.me and MAKE files*
C:\ZORTECH\BIN	*All executable files (.EXE and .COM)*
C:\ZORTECH\LIB	*All library files (.LIB)*
C:\ZORTECH\INCLUDE	*All header files (.H and .HPP)*
C:\ZORTECH\INCLUDE\SYS	*The header file STAT.H*
C:\ZORTECH\UTIL	*Utility programs*
C:\ZORTECH\FG	*Flash Graphics demos*

Before running ZTCSETUP you should make copies of the distribution disks and use these working copies for the installation. You should then put the distribution disks and your working copies in a safe place. To run ZTCSETUP place the working copy of system disk #1 in drive A:. If you are not at the A> prompt type A: and press Enter. At the A> prompt type:

```
ZTCSETUP
```

and press Enter. Follow the displayed instructions. As the files from each disk are copied, you will be prompted for the next disk. You should ensure that you insert the correct disk at each point.

Environment Variables

Your AUTOEXEC.BAT and CONFIG.SYS files will need to be changed. Refer to the *Quick Start* section for details of the changes required. After altering these files, reboot your computer to set up the system environment. If you are happy with this default selection you can now proceed to use Zortech C++. If you wish to alter this default organization you will also need to reconfigure the 'Context Sensitive Help' system and the ZED program editor. You should consult the relevant sections of the manual for an explanation of how this is done.

Altering Directory Paths and Environment Variables

If you wish to alter the default installation of Zortech C++ you will need to understand how Zortech C++ searches directory paths for executable, include, and library files. Each step in Zortech C++ needs to locate various files in order to function properly. For some steps, these files are executables such as the compiler, the linker, or the optimizer. For other steps, these files are source files referenced by the #include preprocessor directive or libraries used by the linker. Directories are searched in a particular order. The highest precedence is given to a directory specification prefixing a filename or following a command line switch. Following that, directory paths defined in Environment Variables are used. The current directory also serves as a default in some cases.

Five environment variables are used to specify directory paths in Zortech C++; PATH, INCLUDE, LIB, TMP and TEMP. The PATH Environment Variable identifies the path to search for executables. It is used in the normal manner for commands entered into the MS-DOS command processor. If no PATH is specified, the default is the current directory.

The INCLUDE Environment Variable identifies the search path for certain files referenced by the #include preprocessor directive. Any file name in a #include directive that contains a directory reference must be found in the specified directory or it will not be found at all.

For files bracketed by the less than (<) and greater than (>) symbols, the compiler searches for the file along the path defined in the INCLUDE Environment Variable. If the file is not found in any of the INCLUDE directories, then that's it - the file is not found. The default directory is NOT searched in this case.

If no INCLUDE Environment Variable has been defined, then only the current directory is searched. Any file bracketed by double quotes will be sought only in the current directory, unless a directory specification has been included within the quotes.

The LIB Environment Variable contains the library search path used by the linker. The run-time library and those library names passed on the command line without a directory specification are sought in the LIB directories. The relationship between the current directory and the directory path defined in LIB is the same as defined above for INCLUDE; if LIB is defined, then the linker will look through only those directories. If LIB is not defined, then only the current directory is used.

The TMP Environment Variable defines where the compiler is to create it's temporary files. If available, it is most efficient if TMP references a RAM disk. If TMP is not defined, then the TEMP variable is used instead. If neither variable is defined, the current directory is used.

For example, if the Environment variables were set as:

```
SET PATH=C:\ZORTECH\BIN
SET INCLUDE=C:\ZORTECH\INCLUDE;C:\SOURCE
SET LIB=C:\ZORTECH\LIB;C:\OTHERLIBS
```

the following examples are true:

```
#include <stdio.h>
```

The compiler will search through C:\ZORTECH\INCLUDE and C:\SOURCE for stdio.h.

```
#include "macros.c"
```

The compiler will search for macros.c in the current directory followed by C:\ZORTECH\INCLUDE\and then C:\SOURCE.

```
ZTC \NEWCODE\SOURCE.C MYLIB.LIB
```

ZTC looks to compile and link \NEWCODE\SOURCE.C. ZTC searches throught the directories C:\ZORTECH\LIB and C:\OTHERLIBS for the library file MYLIB.LIB and also the run-time library. For running ZTC itself and the executables within ZTC, directory lookup will first be done in the current directory and then the directory C:\ZORTECH\BIN. Since no TMP or TEMP Environment Variable was defined, temporary files for the compiler are created in the current directory.

CFLAGS Environment Variable

The Environment Variable CFLAG is used in Zortech C++ to specify command line defaults for ZTC, the one pass compile and link command. If CFLAGS is set to:

```
SET CFLAGS=-w
```

-w is the compiler switch to suppress warning messages.

Warning messages will be suppressed on ZTC commands just as if the -w switch had been included on the command line.

Floppy Disk Installation

Although it is *possible* to install Zortech C++ on a floppy based system (if you have sufficient experience) we do not recommend it. However, for those who are determined we offer the following outline:

It is suggested that the compiler files together with the system header files be placed on a "COMPILER" disk. The linker and the library files for the memory model being used should be placed on a separate "LINKER" disk, and finally the program editor ZED and your source files should be placed on a third "DEVELOPMENT" disk. This arrangement will inevitably involve swapping between the compiler and linker disks in drive A: whilst your development disk remains in drive B:

CONTEXT SENSITIVE HELP

Introduction

This software provides a "Context Sensitive Help" feature for ZORTECH C and C++. It is a memory resident program written in Zortech C with the Zortech Hotkey Toolkit.

When you are editing programs and feel the need to reach for your manual, think twice. Simply position the cursor on the command or function you need help with and press the hotkey. The resident helper will open a window on the screen, taking care not to cover the offending function, and present you with the appropriate help pages straight from the manual.

You can then browse through the information and even ask for further help on any related item. When you have had your memory suitably jogged, you can return to your editor and finish the program, knowing that help is just a keypress away.

Configuring the Software

If you have used the ZTCSETUP program to install Zortech C++ on your hard disk you will have installed a pre-configured copy of ZTCHELP.COM in the \ZORTECH\BIN directory. The default hotkey is <ALT><H>. You do not need to reconfigure this program unless the default setup is unsatisfactory.

If you wish to change the default configuration you will need to use the program ZTCHCONF.EXE which is on Disk No. 5. If you run this and answer the questions it will create a tailored version of ZTCHELP.COM, which is the resident help software. Although at no time during configuration is the uninstalled version of your software at risk, it is always good practice to secure all disks before use!

The items you are allowed to tailor are;

HOTKEY What key combination do you prefer.

INTERRUPT Software needs access to an unused interrupt. You may select from 0x70 to 0x7F.

SCREEN CHARACTERISTICS

Information about your screen.Available options are:

1: Let the software detect screen type
2: Default Color
3: Default Mono
4: Specify your own values
Q: Quit

FILENAMES You can choose where to locate the help files, ZTC.HLP, ZTC.HDX, and the scratch file TEMP.FIL.

After configuring the help system you should copy the new ZTCHELP. COM file into a directory which is on the directory search PATH.

Installing the Software

Omit this section if you have installed the help system using ZTCSETUP.

To run the help system, you need the help files ZTC.HLP and ZTC.HDX, and a configured version of the resident software (ZTCHELP.COM). You do not need ZTCHCONF.EXE or ZTCHELP.UNI. These can remain on your distribution disk, you will need them if you ever change the configuration. Copy ZTC.HLP and ZTC.HDX to the location you chose. Then copy ZTCHELP.COM to whatever directory you require it in.

NOTE: If you used ZTCSETUP.EXE to install Zortech C++, the help program was transferred to your hard disk in the \ZORTECH\BIN directory.

You may decide to load ZTCHELP in your AUTOEXEC.BAT file or only before a ZTC session, the choice is yours.

Loading the Resident Helper

If the help system is not installed automatically in your AUTOEXEC.BAT file, it can be installed memory resident at any time. To load the memory resident software, simply enter ZTCHELP. An installation message will be displayed to confirm the hot key and screen type. If you try to reload the software you will be told that it is already resident. After the installed message is displayed, the DOS prompt will reappear and you can then continue to load ZED and edit your C source programs.

Removing the Resident Helper

Provided ZTCHELP was the last memory resident program to be installed it can be un-installed by entering the following command at the DOS prompt:

```
ZTCHELP /r
```

(Presuming ZTCHELP.COM is in the current directory or in a directory included in the PATH command.)

Invoking Help

When you require help, simply place the cursor on or near the function you require help on, and press your chosen hot-key combination. A window will open with the relevant help on display. If no help can be found for the function pointed at, you will be given the help topic nearest to the word you are searching for.

If no similar match can be found you will be given the index screens which allow you to browse through the index and find the relevant help. Once in the help system you can use the following keys to navigate your way around.

Keys Summary:

Cursor Left/Right/Up/Down	}	navigation through help
Page up/down	}	and index
Home/End	}	
Enter	}	Seek another level of help
Esc	}	When in help file, Restore screen -exit When in index returns to previous level, either application or help file.
F1	}	When in help file Just exit- leave window
F10		When in help file calls Key usage window.
F9	}	Jumps through Topics
?	}	When in help file, calls Index.

ZED EDITING ENVIRONMENT

ZED is an integrated editor/compiler environment, which allows you to edit C and C++ programs, and compile them within the editor. This frees the programmer from the edit /save /compile /edit cycle, dramatically improving productivity.

ZED is easy to use for the beginner, but has powerful features to suit the experienced programmer. Those who regularly use Wordstar (TM) will find that most Wordstar commands will also work in ZED.

ZED will edit up to 5 files at a time, and the file(s) may be as large as you like (subject only to the amount of memory you have installed in your machine).

ZED will run on IBM (TM) PC/XT/AT or compatibles, running DOS v2.10 or later, and requires an absolute minimum of 256k memory (more memory is required to edit more than 1 file, or to compile within the editor).

If you have used ZTCSETUP, ZED will have been installed with a default configuration suitable for use with the directory structure used by the setup program as described in the *Detailed Installation Guide.* If you wish to customize ZED, please refer to the later section on *Configuration.*

You will be able to learn how to use ZED without having to constantly refer to the manual. Skim through the manual quickly, to get some idea of the sort of things ZED can do, then try it. Use the help screens (press F1) and menus until you find they are slowing you down, then refer to the menu map and/or alternate key list for shorthand ways of doing things. Most edit functions can be accomplished with one or two keystrokes, but, until you learn the quick keystrokes, just press Escape to see the main command menu.

Starting ZED

From your work directory, type ZED followed by the file name of the file you wish to edit (e.g. ZED myprog.cpp). ZED will accept full pathnames, so you are not limited to editing files in the current directory.

The screen will look like this :

```
ZED - Zortech Editor - Version 2.60                Press Escape for command menu
    0:   1:  1   INSERT                                          c:\myprog.c 1

        /*
         *      THE TEXT OF YOUR PROGRAM HERE
        /*
```

The first line tells you the version number of your copy of ZED, and reminds you that the command menu is accessed by pressing Escape. This line is also used for the menu line.

The second line is the status line, which is used to display editing iformation.

```
0:   1:  1   INSERT                                    c:\myprog.c 1
```

The first figure is the total number of charcters typed so far, the next two figures are the row and column position of the cursor. The next space is replaced by a number when a keyboard macro is being recorded, or by a 'P' if a macro is being played back.

INSERT means that anything you type will be inserted at the cursor position. **OVERTYPE** in this position means that typed text will overwrite any text underneath.

AUTOINDENT can appear here when it is toggled on (using Shift Tab).

The last two fields are the name of the file being edited, and the ZED buffer number (ZED has 5 buffers, allowing you to edit up to 5 files at the same time).

The third line is the ruler line, with the arrows representing tab positions.

You may also execute ZED without any parameters, in which case the file name will be "no file". When "no file" is shown, ZED allows you to edit its memory buffers if you wish, but you should normally use the EDIT command (Esc F E or Alt E) so that ZED can keep automatic backups for you.

WARNING

If you use the Read command to read a file into the memory buffer and the Write command to write the buffer back to the disk, ZED will not keep automatic backups for you, and you could easily overwrite a valuable source file by mistake.

The Write and Read commands are intended to write sections of the buffer to a disk file, and read them back into another file, or to keep a series of standard functions for use with slight changes in multiple source files.

Key Bindings

In all the following command descriptions, key sequences to access the listed commands are given in brackets. For example, (Esc E R) means press Esc (to get to the main menu), followed by E (for the Edit menu) and R (for the Restore function). Note also that you sometimes need to press key combinations. Key combinations are written as follows :-

^ Q	Press the Q key while holding down the Control key.
^ Break	Press the Break (often marked "Scroll Lock") while holding down the Control key.
Alt + X	Press the X key while holding down the Alt key.
Shift + Tab	Press the Tab key while holding down the Shift key.

Function Keys

F1 HELP-Press at any time for context-sensitive help.

F2 UNDO - Recalls the last text you deleted. ZED remembers up to 100 items deleted (max 10000 characters). *(Esc E R)*

F3 Insert saved block buffer at cursor. *(Esc B R)*

F4 Delete text between cursor and TAG, place in saved block buffer. *(Esc B S)*

F5 Exchange TAG and cursor. *(Esc M T)*

F6 Place TAG at cursor position. *(Esc B T)*

F7 Search forwards (for the text string you have set up with Esc S F) *(Esc S N)*

F8 Search backwards *(Esc S L)*

F9 Delete from cursor to the end of the current line. If the cursor is at the start of a line, this deletes the whole line and moves subsequent lines up to fill the space. If the cursor is at the end of a line, joins this line to the next one. *(Esc D E)*

F10 Delete word. If the cursor is on a space, this just deletes all spaces until the start of the next word (or the end of the line). *(Esc D W)*

You can see a list of the function key assignments by pressing F1 while editing. All these functions (except HELP) can also be performed by using the menus - the menu selections are shown in brackets after each command. You may also use ZCONFIG to change these key assignments if you wish.

Cursor Keys

The cursor keypad is used as follows :

Left arrow	Moves the cursor left, unless it is in column 1.
Right arrow	Moves the cursor right, unless it is at the end of a line.
Up arrow	Moves the cursor up 1 line.
Down arrow	Moves the cursor down 1 line.
Home	Moves the cursor to the start of the current line. (Esc M L)
End	Moves the cursor to the end of the current line. (Esc M R)
Ctrl + Left	Moves left one word.
Ctrl + Right	Moves right one word.
PgUp	Moves the cursor up 10 lines (number of lines can be changed - see section on ZCONFIG)
PgDn	Moves the cursor down 10 lines (number of lines can be changed - see section on ZCONFIG)
Ctrl + Home	Moves to the top of the screen. (Esc M P)
Ctrl + End	Moves to the bottom of the screen. (Esc M M)

Ctrl + PgUp	Moves to the start of the file.(Esc M S)
Ctrl + PgDn	Moves to the end of the file.(Esc M E)
Ins	Toggles between Insert & Overtype mode
Del	Deletes the character under the cursor.
Backspace	If in INSERT mode, deletes the character to the left of the cursor. In OVERTYPE mode, just moves the cursor left.
Shift + Tab	Toggles Auto-indent mode ON/OFF (Esc E A)

Note that many of these functions can also be performed by using the menus - the menu selections are shown in brackets after each command.

You may also use ZCONFIG to change these key assignments if you wish.

Wordstar Compatible Keys

(Note - ^ means hold down the control key and press the following key)

Note that some of these functions can also be performed by using the menus . You may also use ZCONFIG to change these key assignments if you wish.

^S	Char left
^A	Word left
^G	Delete char
^L	Find/replace again
^D	Char right
^F	Word right
^T	Delete word
^Q	Quick move menu
^E	Line up
^R	Page up
^Y	Delete Line
^K	Block file menu
^X	Line down
^C	Page down
^V	Toggle insert
^B	Format part of the buffer

Other Control Keys

The remaining control keys generate the same key codes as other keys on the keyboard.

^F1 to ^F10

Record key macros on the appropriate function key. Macro recording can also be done via the Record menu (Esc R 1) to (Esc R 0). You may use ZCONFIG to change these key assignments if you wish.

^ Break

Interrupts whatever you are doing. You may press ^ Break at any time. When you do so, the system will respond with the warning message "Interrupted-press Esc". Pressing Esc will take you straight back into edit mode.

If you press ^ Break while saving a file, the file will NOT be saved to disk. If you interrupt a file load or edit when part of the file has been read in, that part of the file will remain in the text buffer, but the file name will be displayed as "no file", and any attempt to save the file will prompt you for a new file name. This prevents you accidentally truncating the file.

Alternate Keys

Alt + A	search for string ignoring cAse (Esc S C)
Alt + B	move to Block start marker (Esc M B)
Alt + C	Compile (Esc O O)
Alt + D	DOS command (Esc O D)
Alt + E	Edit file (Esc F E)
Alt + F	Find string (Esc S F)
Alt + G	Global replace found string (Esc S G)
Alt + H	**Reserved for memory resident help.**
Alt + I	Input (read) file into text at cursor (Esc F R)
Alt + J	move cursor to corresponding bracket/end of block (Esc M [)
Alt + K	move to blocK end marker (Esc M K)
Alt + L	set Left margin (Esc T L)
Alt + M	set right Margin (Esc T R)
Alt + N	move directly to a line Number in the file (Esc M N)
Alt + O	cOmpare two buffers (Esc O C)

Alt + P	Print part of buffer (Esc O P)
Alt + Q	Quit current buffer, abandoning any changes (Esc F A)
Alt + R	Replace found string (Esc F R)
Alt + S	Save current version of file to disk (Esc F S)
Alt + T	set Tab length (Esc T T)
Alt + U	right-jUstify C comment (Esc E C)
Alt + V	saVe current buffer and exit the editor (^ K X).
Alt + W	Write part of buffer to file (Esc F W)
Alt + X	Exit to DOS (Esc F X)
Alt + Y	mark start of block (Esc B B)
Alt + Z	mark end of block (Esc B K)
Alt + 1	edit buffer 1
Alt + 2	edit buffer 2
Alt + 3	edit buffer 3
Alt + 4	edit buffer 4
Alt + 5	edit buffer 5
Alt + 6	not used

Alt + 7	not used
Alt + 8	not used
Alt + 9	move text left 1 tab stop (Esc E -)
Alt + 0	move text right 1 tab stop (Esc E +)

All of these functions can be performed using the menus - the menu selections are shown in brackets after each command. In the above descriptions, the capital letter indicates why a particular alternate key has been chosen for a command.

Note that these key assignments are just the defaults, as used by the author of ZED. You may use ZCONFIG to change them if you wish.

We suggest that as you become familiar with ZED you use ZCONFIG to set up as many keys as you can remember to do those commands you use most often, so that you only need to use the menus occasionally.

Alt + F1 to Alt + F10 Play back macros recorded using ^ F1 to ^ F10. You can also use the Playback menu (Esc P 1) to (Esc P 0). You may use ZCONFIG to change these key assignments if you wish.

Online Help

When you press F1, a context-sensitive help screen appears. This help screen will usually contain at least one cross-reference option, which will be highlighted. To see other pages of help, move the cursor to the cross-reference page you require (using cursor keys, Home & End), and press Return.

When you have read the help screen(s), press Escape or F1 to return to what you were doing before. You may change the help text by editing the file ZED.HLP (See HELP FILE under CONFIGURATION below).

This help is in addition to the memory resident C++ help. Which can be called up at any time simply by pressing Alt, and which covers the C++ syntax and library functions.

Menus

ZED has two kinds of menus.

Bar menus appear as a list of options at the top of the screen, with the first option highlighted. An explanation of that option is shown on line 2 (often just a list of the options on a sub-menu). To select the highlighted option, press Return.

To move the highlight, use the cursor left & right keys, or space & backspace. You will notice that each option has only one capital letter (usually the first letter) - you may select an option directly (without first highlighting it) by pressing that letter key.

You may also return from any menu without selecting an option by pressing Escape. Notice also that those function, cursor, control and alternate keys which only move the cursor or alter the text still work, even from within a bar menu.

Window menus pop down in a window covering part of the text you are editing. They have one option per line, with the explanation appearing on the last two lines of the window. To select the highlighted option, press Return. To move the highlight, use the cursor up & down keys, or space & backspace.

Like bar menus, each option has only one capital letter (usually the first letter) - you may select an option directly (without first highlighting it) by pressing that letter key. You may also return from any menu without selecting an option by pressing Escape. The function, cursor, control and alternate keys do not work from within a window menu.

You may use ZCONFIG to position window menus anywhere on the screen, or to change window menus into bar menus (and vice versa). You will find that, although window menus give more detailed option names, bar menus work faster, and don't restrict cursor movement. Once you are familiar with all the menus, you may find it speeds your use of ZED if you change all the menus to bar menus.

Main Menu

The ZED main menu is accessed by pressing Escape. The main menu options are as follows :

File	To save the current file, edit new file(s), read and write bits of files, etc.
Move	To quickly move the cursor round the file.
Edit	To change mode, or do useful editing operations like UNDO, change text to upper/lower case etc.
Blocks	To tag, delete, copy or move blocks of text, or place invisible markers in the text to move back to later.
Delete	To delete text.
Search	Search for strings and optionally replace with another string.
Other	Miscellaneous commands - Compile, Link, Compare files, Print, other DOS commands, saving your favorite editor settings permanently to disk.
Record	Record keystrokes in a key macro or file. Save key macros to disk.
Playback	Playback keystrokes recorded with Record.

Text Setting tabs and margins, formatting paragraphs, and other text layout commands.

1 - 5 Change to buffer 1,2,3,4 or 5. Each of the 5 edit buffers can contain a separate file, so you can edit a number of interdependent source files at the same time. ZED remembers the cursor position, and all the various marker positions, in each buffer. There is also a command to compare two buffers, which is very useful for updating similar source files in the same way, or for finding out what changes have been made to the latest version of a source.

Files Menu

When a file has been edited, the pre-edition file is kept as a backup. Normally the backup file has the same name as the original, but with an extension of ".BAK". You may use ZCONFIG to tell ZED to keep all backup files in one directory (so you can easily delete them if you run out of disk space), in which case the backup will have the same name as the original, but be on the specified directory. This option is particularly handy for C sources where header files (.h) and source files (. c) have the same name but different extensions, as the full file name and extension is retained in the backup file.

Edit *(Alt + E)* Saves the current file (if any), then prompts for a new file to edit. You may use a full pathname if required. See below for details on entering file names. You should always use Edit whether you want to read an existing file into memory for editing, or you want to create a new file.

Save *(Alt + S)* Save any changes made so far to disk, and continue editing the file. This command is the one you wish you had done more often when you have a power cut ! We suggest you use it fairly frequently when you are adding lots of text to a file, just in case. This command acts as if you had saved the file, and re-edited it, so a backup copy will be created, over-writing any previous backup. This command does not affect the cursor position.

Abort *(Alt+Q)* Abandon any changes made to the current buffer since the last save, and clear the buffer. If you have changed the buffer, Abort will ask for confirmation before losing all those changes.

Read *(Alt + I)* Read the contents of a file at the cursor position. This command is intended for including one file inside another, or for making a new file which is based on an existing file with a different name.

Write *(Alt + W)* Write part of the current buffer to a disk file. This command is handy when your source file becomes unwieldy, and you wish to split into different sections, or when you wish to extract a few functions from a source file. It is also useful if you decide you want to leave your original file unchanged, but keep the edited version under another file name.

When you have selected Write, and entered the name of the file to write, you are presented with the Size menu, so you can decide what part of the buffer to write to the file.

eXit *(Alt + X)* Attempts to leave the editor. If any buffers have been edited since the last save, displays each one and asks if you want to save it. An answer of 'y' or Return will save the buffer, and continue. An answer of 'q' will abandon any changes you have made to that buffer, and continue. Any other answer returns to editing. If no buffers were changed, or you answered 'y' or Return or 'q' to all the questions, leaves the editor (returning to DOS).

1-5 *(Alt + 1* to *Alt + 5)* Change buffer number.

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG.

When ZED is loading a file, it displays the first screenful of the file as soon as it is loaded, but carries on loading until the whole file is in memory. The message "Loading file ", followed by the file name, is displayed during loading. If you change your mind part way through loading a long file, you may press Control-Break to interrupt the load. If you do this, part of the file will be left in the buffer.

Entering file names

The last file name used is presented as the default. To use this name, press Enter. You may also edit the name in a similar way to buffer editing, as follows :-

F1	Display help.
Left arrow	Moves cursor left 1 character.
^ Left arrow	Moves the cursor left 1 word.
Right arrow	Moves cursor right 1 character.
^ Right arrow	Moves the cursor right 1 word.
Home	Moves cursor to beginning of string.
End	Moves cursor to end of string.
Del	Deletes the character under the cursor.
^ T or F10	Deletes the next word.
F9	Deletes from the cursor to the end of the line.
^ Y	Deletes the whole of the line.
Return	Accepts the string.

Ins	Toggles Insert mode - In insert mode, the cursor changes to a flashing underline. Anything you type will be inserted, pushing remaining characters to the right. In normal (overtype) mode, the cursor changes to a highlighted block. Anything you type will replace the character underneath the cursor.
Esc	Abort the current command.
Backspace	If the cursor is at the end of the string, or you are in insert mode (the cursor is a flashing underline), Backspace deletes the character to the left of the cursor. Otherwise, Backspace just moves the cursor left 1 character.

Note that if you wish to type a completely different string, press F9 or ^Y first to clear the default. If you forget to do this, and the old string is longer than the new one, you should press F9 before pressing Return, to clear the last few characters left from the old string.

At any time when you are prompted for a file name, you may enter an ambiguous name (e.g. *.* or \include*.h etc.). A list of all the files satisfying the file name will be shown on the screen. You can then use the cursor keys to move around the list, and press Return to select the file you want. You can also press Escape to take you back to the filename prompt, to enter a different filename.

Size menu

The Write function operates on part of the buffer - the size menu allows you to select which part :-

Word	From the cursor position to the end of the next word.
End of line	From the cursor position to the end of the current line.
Line	The whole of the current line.
Paragraph	From the cursor position until the next completely empty line.
Block	The marked block.
Tag	Everything between the cursor and the TAG marker.
}	Everything between the cursor and the next corresponding '}'.
All buffer	The whole buffer.
No of lines	Specify how many lines to write.

Moves Menu

Left *(Home)* Moves the cursor to the start of the current line.

toP *(^ Home)* Moves to the top of the screen.

Start *(^ PgUp)* Moves to the start of the file.

Right *(End)* Moves to the end of the current line.

bottoM *(^ End)* Moves to the bottom of the screen.

End *(^ PgDn)* Moves to the end of the file.

Number *(Alt + N)* Prompts for a line number, and then moves directly to that line.

Tag (F5) Exchanges the cursor and the TAG marker. Each ZED buffer has a TAG marker, which can be used to move back and forth between two places in the buffer - when you press F5 (or Esc M T), the TAG and the cursor are exchanged, so pressing F5 (or Esc M T) again returns you to where you were. The TAG marker can also be used as a block delimiter (all text between the cursor and the TAG counts as the block). To set the TAG marker use (F6), or see the Blocks menu.

[*(Alt + J)* If cursor is on a bracket (,) , [,], { or }, moves the cursor to the corresponding close bracket. Otherwise moves the cursor to the end of the current program block (delimited by { and }).

This facility is useful in determining when a variable goes out of scope. Simply place the cursor on the variable declaration and press (Alt + J).

Block *(Alt + B)* Moves to the block start marker. To set the block start marker use (Alt + Y), or see the Blocks menu.

blocK *(Alt + K)* Moves to the block end marker. To set the block end marker use (Alt + Z), or see the Blocks menu.

1 - 9 Moves to invisible numbered marker 1 to 9. To set the marker, see the Blocks menu. Note that moving to a marker is the only way of finding out where it is.

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG.

Edit Menu

Insert Enter Insert mode. Any characters you type will be inserted at the cursor position. Pressing Enter inserts a new line. You can also use the (Ins) key, or (^V) to toggle between Insert and Overtype mode. ZED remembers which mode it was in last time it was used.

Overtyp Enter Overtype mode. Any characters you type will replace the character at the cursor, unless you are at the end of a line, when they will be added to the line. Pressing Enter just moves the cursor to the next line. You can also use the (Ins) key, or (^ V) to toggle between Insert and Overtype mode.

Rest *(F2)* Restore the last item deleted at the cursor position. The editor remembers up to 100 items deleted (to 10000 characters), and you can get them all back by repeatedly pressing F2. This command is useful for moving small amounts of text around the buffer (just delete it, then move to where you want it, then restore the deletion), as well as for recovering from mistakes.

Lower Convert part of the buffer to lower case. When you select this command you are presented with the size menu, so you can decide how much of the buffer to convert to lower case. The default option is Word - the rest of the current word.

Upper Convert part of the buffer to upper case. When you select this command you are presented with the size menu, so you can decide how much of the buffer to convert to upper case. The default option is Word - the rest of the current word.

Graphic Insert a specific ASCII graphic character at the cursor. This is used to insert graphics characters, and printer control characters. You may insert any ASCII character except the following :-

0	Null characters are not allowed in ZED
9	This is the TAB character - use the TAB key for this
10	Line feed character - delimits end of line
13	Return character - delimits end ofline
26	EOF character - delimits end of file

See your printer manual for details of the control codes for your printer.

Setseq Initialize the sequence number. See seQuence.

seQuenc Insert the current value of the sequence number in the text, and increment the sequence number. The two sequence commands are useful if you want a list of consecutive numbers (e.g. a list of #defines). First set the sequence number to the first value you want (e.g. 1) using (Esc E S). You can then set up a macro with (for instance) the following keys (End Esc E Q Down), and that will insert consecutive numbers at the end of each line each time you execute it.

Comment *(Alt + U)* Right-justify and terminate the C comment on the current line. This command makes it easy for you to line up all your comments on the right hand margin, so they look nice and tidy. It only works on one-line comments. You don't even have to type the closing */, the command will supply it automatically if there isn't one. If there is no open comment /* on the current line, the command does nothing. If you haven't set a right margin (using Esc T R), the command assumes a default of 79.

+ *(Alt + 0)* Moves part of the buffer right 1 tab stop. When you select this command you are presented with the size menu, so you can decide how much of the buffer to indent. The default option is [- the rest of the current C block (terminated by }).

- *(Alt + 9)* Moves part of the buffer left 1 tab stop. When you select this command you are presented with the size menu, so you can decide how much of the buffer to "un-indent". The default option is [- the rest of the current C block (terminated by }). These commands are useful for correctly indenting blocks of C code when you add a new pair of braces around some existing code.

The last two commands can be interrupted with Control-Break.

Auto *(Shift Tab)* Toggles auto-indent on/off. Auto-indent inserts spaces and tabs at the start of each new line, so that it lines up with the line above. This is useful when you are writing some new code within a set of braces, so that it lines up neatly to show program logic. Autoindent only adds the spaces and/or tabs each time you press the < Return > key. If you just type continuous text (like this manual) using word-wrap, autoindent does not occur. ZED remembers the autoindent setting from the last time it was used.

1-5 Change buffer number.

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG.

Size menu :

The Lower, Upper, + and - functions operate on part of the buffer - the size menu allows you to select which part :-

Word	From the cursor position to the end of the next word.
End of line	From the cursor position to the end of the current line.
Line	The whole of the current line.
Paragraph	From the cursor position until the next completely empty line.
Block	The marked block.
Tag	Everything between the cursor and the TAG marker.
]	Everything between the cursor and the next corresponding '}'.
All buffer	The whole buffer.
No of lines	Specify how many lines to convert/indent.

Blocks Menu

Block *(Alt + Y)* Place the block start marker at the cursor.

blocK *(Alt + Z)* Place the block end marker at the cursor. These block markers are used as an alternative to the TAG for marking an area of text. The TAG is faster, using the TAG and the cursor as the block delimiters, but you can use the block markers if you are more comfortable with them. Normally, when both block markers have been placed in the text, the block is highlighted. You can turn this off if you wish (See Hilite below).

Delete Deletes the currently marked block, and places it in the block holding buffer. The block holding buffer is used by all block transfers, and always contains a copy of the last block you deleted, copied, moved or yanked. You must have first marked your block, using (Esc B B) and (Esc B K).

Move Deletes the currently marked block, and places it in the block holding buffer, then copies the block holding buffer back into the text at the cursor. Before this command will work, you first need to have marked your block, using (Esc B B) and (Esc B K).

Copy Copies the currently marked block into the block holding buffer, then copies the block holding buffer back into the text at the cursor. Before this will work, you must have first marked your block, using (Esc B B) and (Esc B K). If you need more than one copy, either repeat this command, or just press F3 for each subsequent copy.

Save *(F4)* Deletes all text between the cursor and the TAG marker, and places it in the block holding buffer. This is the quickest way to copy or move blocks of text, as follows : - Cursor to one end of the block and press F6 - Tag at cursor. Cursor to other end and press F4 - the block disappears. If you wish to copy the block (rather than move it), press F3 immediately - the block re-appears. Cursor to destination and press F3 - the block re-appears. If you need more than 1 copy, just keep pressing F3.

Restore *(F3)* Copies the block holding buffer back into the text at the cursor.

Tag *(F6)* Places the TAG marker at the cursor position. Note that F5 swaps the TAG and the cursor, so you can see where the TAG is.

Hilite Toggles display of current block in inverse video. Normally, when both block markers have been placed in the text, the block is shown highlighted. This command turns this highlighting on or off. ZED remembers whether highlighting was used from the last time it was used.

Yank Yanks (copies) part of the text into the block holding buffer, without altering the text. Takes you into the size menu to decide how much text to yank. You can insert the text any-where in any buffer at the cursor position by pressing F3 (or Esc B R). This command is particularly useful for moving blocks of text from one file to another without altering the source file.

1-9 Places an invisible marker (number 1-9) at the cursor position. You can move back to this marker using the corresponding marker in the Move menu (Esc M 1) to (Esc M 9).

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG.

Delete Menu

You may delete any part of the buffer. Which part depends on your response to the the size menu :

Word	(F10) From the cursor position to the end of the next word.
End of line	(F9) From the cursor position to the end of the current line.
Line	(^Y) The whole of the current line.
Paragraph	From the cursor position until the next completely blank line.
Block	The marked block.
Tag	Everything between the cursor and the TAG marker.
]	Everything between the cursor and the next corresponding '}'.
All buffer	Deletes the whole buffer (CAREFUL - you may not be able to UNDO this)
No of lines	Specify how many lines to delete (starting at the cursor line). You may not be able to UNDO this if you delete too many.

Quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG. Everything you delete is saved in the UNDO stack, and may be recalled by pressing F2 or (Esc E R).

If any block larger than 10,000 characters is deleted then it cannot be recalled.

Search Menu

Find *(Alt + F)* Set up a string to search for. ZED remembers this string until you change it, so repeated searches are easy. This command only sets up the string, to actually search for it, you will need to press F7 (search forwards) or F8 (search backwards)

Case *(Alt + A)* Set up a string to look for (ignore upper/lower case distinctions). ZED remembers this string until you change it, so repeated searches are easy. This command only sets up the string, to actually search for it, you will need to press F7 (search forwards) or F8 (search backwards)

Next *(F7)* Find next occurrence (search forwards). Before using this command, you need to set up the string using the Find or Case commands above.

Last *(F8)* Find previous occurrence (search backwards). Before using this command, you need to set up the string using the Find or Case commands above.

Replace *(Alt + R)* Replace found item with another string. You will be asked for the replacement string. If the cursor is already at an occurrence of the search string, it will be replaced immediately. Otherwise press F7 or F8 to find the search string, and you will be asked whether you wish to replace it. Press y (or Enter) to replace, or any other key to leave unchanged.

Global *(Alt + G)* Replace throughout rest of file (automatically without asking). You will be asked for the replacement string. If the cursor is already at an occurrence of the search string, it will be replaced immediately, as will all the rest of the occurrences from the cursor to the end or beginning of the file (depending on whether your last search was forwards or backwards). Otherwise press F7 or F8 to find the search string, and you will be asked whether you really wish to replace globally. Press y (or Enter) to replace, or any other key to leave the file unchanged.

eXit Return to editing.

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG. If you are familiar with Wordstar, you can still use Wordstar commands for search and replace (^ Q F or ^ Q A). The Search commands may be interrupted with Control-Break.

Entering strings

The last string used is presented as the default.

To use this, press Enter. You may also edit the string in a similar way to buffer editing, as follows:

F1	Display help.
Left arrow	Moves cursor left 1 character.
^ Left arrow	Moves the cursor left 1 word.
Right arrow	Moves cursor right 1 character.
^ Right arrow	Moves the cursor right 1 word.
Home	Moves cursor to beginning of string.
End	Moves cursor to end of string.
Del	Deletes the character under the cursor.
^ T or F10	Deletes the next word.
F9	Deletes from the cursor to the end of the line.
^ Y	Deletes the whole of the line.
Return	Accepts the string as displayed (no matter where the cursor is)

Ins Toggles Insert mode - In insert mode, the cursor changes to flashing or underline. Anything you type will be inserted, pushing remaining characters to the right. In normal (overtype) mode, the cursor changes to a highlighted block. Anything you type will replace the character underneath the cursor.

Esc Abort the current command.

Backspace If the cursor is at the end of the string, or you are in insert mode (the cursor is flashing or underline), Backspace deletes the character to the left of the cursor. Otherwise, Backspace just moves the cursor left 1 character.

Note : If you wish to type a completely different string, press F9 or ^ Y first to clear the default. If you forget to do this, and the old string is longer than the new one, you should press F9 before pressing Return, to clear the last few characters left from the old string.

Other Menu

cOmpile *(Alt + C)* Compile the program in the buffer - You can only compile files with extensions ".c" or ".cpp". For more details on compiling, see below.

Compare *(Alt + O)* Enables you to compare the contents of any two buffers. Just select a buffer number with which to compare the current buffer. The comparison starts at the current cursor position in each buffer, and stops at the first difference. If you have noted the difference, and want to continue, just move the cursor in each buffer to pass over the difference, so that the two cursors end up over identical characters, and Compare again. The Compare command may be interrupted with Control-Break.

DOS *(Alt + D)* Run a DOS program, and return to editing afterwards. You may even run a copy of the DOS command processor (so you get an A > or C > prompt) by entering an empty program name (run DOS command EXIT to get back to the editor if you do this). This command will not work on DOS versions earlier than 2.10. If you are running DOS 2 or 3, and you still have trouble with this command, see the note on COMSPEC below.

Print *(Alt + P)* Print part of the buffer - the size menu selects which part of the buffer to print. The Print command may be interrupted with Control-Break.

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG.

Entering strings

The last string used is presented as the default.

To use this, press Enter. You may also edit the string in a similar way to buffer editing, as follows:

F1	Display help.
Left arrow	Moves cursor left 1 character.
^ Left arrow	Moves the cursor left 1 word.
Right arrow	Moves cursor right 1 character.
^ Right arrow	Moves the cursor right 1 word.
Home	Moves cursor to beginning of string.
End	Moves cursor to end of string.
Del	Deletes the character under the cursor.
^ T or F10	Deletes the next word.
F9	Deletes from the cursor to the end of the line.
^ Y	Deletes the whole of the line.
Return	Accepts the string as displayed (no matter where the cursor is)

Ins Toggles Insert mode - In insert mode, the cursor changes to flashing or underline. Anything you type will be inserted, pushing remaining characters to the right. In normal (overtype) mode, the cursor changes to a highlighted block. Anything you type will replace the character underneath the cursor.

Esc Abort the current command.

Backspace If the cursor is at the end of the string, or you are in insert mode (the cursor is flashing or underline), Backspace deletes the character to the left of the cursor. Otherwise, Backspace just moves the cursor left 1 character.

Note : If you wish to type a completely different string, press F9 or ^ Y first to clear the default. If you forget to do this, and the old string is longer than the new one, you should press F9 before pressing Return, to clear the last few characters left from the old string.

Compiling

Once you have selected the Compile option, you may enter any compiler switches. For most programs, you may not need any switches at all, so you can just press return. For full details of the available switches consult the compiler section. Briefly the switches are :

-b	Use Big compiler
-v	Verbose output from compiler
-c	Compile only (don't link)
-r	Generate ROM-able code
-did=tx	#define id tx
-s	Include Stack overflow checking
-e	Show Effect of pre-processor
-u	Suppress aUtomatic macros
-llist	Generate List file list.
-w	Suppress Warning errors
-ipath	Search for Include files on path
-p	Turn off auto-prototyping
-O	optimize - takes a lot longer.
-O-sw	Turn optimizer switch off (eg. -O-dc)
-O + sw	Turn optimizer switch on (eg. -O + space)

-f	Generate inline 8087 code
-ofile	Output to file
-j	Asian language characters
-a	No Alignment restrictions in structures
-J	chars default to unsigned
-C	Include Codeview info in EXE file
-B	Run BUNCH on linker output (MUST use this when linking C ++ file with MSLINK
-g	Insert Line no. info in obj file and generate a map file
-kfile	After compiling the program (but not linking), run MAKE using file as the project file (default file is makefile)
-Kfile	Like -k, but don't compile the program first.
-m <m>	Set Model - < m > may be one of :

S	Small model
SI	Small model Integer arithmetic only
I	Small model Integer arithmetic only
T	Tiny model (COM file)
TI	Tiny model Integer arithmetic only
M	Medium model
MI	Medium model integer arithmetic only
C	Compact model
CI	Compact integer arithmetic only
L	Large model
LI	Large model Integer arithmetic only

Default model is S

If you have not set the -c, -k or -K switches (ie. you wish to Link the program as well as compiling it) you may include other object files names and libraries by entering them here (separated by spaces). The compiler switches are remembered automatically the next time you invoke ZED.

ZED then runs ZTC1 or ZTCPP1 (which must be somewhere on your PATH), with its input coming directly from memory. This will speed up compilation while the syntax of your program is checked.

You can increase the speed even further if you have a Ramdisk, by telling ZED about it with ZCONFIG - it will then be used for the temporary file generated by the compiler.

The compiler is stopped at the first error, and the cursor is placed near the error. If it is a Warning, you may press Enter to continue with the compilation, or Esc to abort. If it is a fatal error, any key will abort the compilation. If no fatal errors occurred, ZED will save your source file to disk, and then automatically run ZTG (if required), ZTC2, LINK, BUNCH and EXE2BIN or MAKE (if required).

If you wish to keep your Zortech library files or include files in different directories to those used by ZTC setup, you will need to tell ZED which directories to use by running ZCONFIG.

Compiling Projects With Multiple Source Files

If you are compiling a project which consists of many inter-related source files, you can simplify the process by creating a project file, and using the supplied MAKE command. If you have only one project per subdirectory (like most people), you should call your project file MAKEFILE, which is the default project file name.

Full details of the MAKE utility, and project files, are given in the MAKE section of this manual. If you have created a project file, you can compile the current program, then bring the whole project up to date with a single command. To do this, include the -k switch in your compile flags. You can edit and compile source files written in other languages (e.g. assembler) by using the -K switch.

Note that the -k and -K switches are only valid for compiling within ZED using the (Esc O O) or (Alt + C) command. They will not be accepted by ZTC - to run MAKE directly from DOS, just type MAKE at the DOS prompt !

Testing & Running DOS Programs

Having linked your program, you will probably want to test it. Just enter any parameters required to the "Program parameters" prompt, and ZED will run the program, which will return straight back into the editor.

Note that if you use the -k or -K switches, ZED will not automatically run your program. You may either run your program by using the DOS command (Esc O D), or by including a program invocation in your makefile.

If you do not want to run your program immediately, press Esc. You may, of course, run your program (or any other DOS program) at any time, without having to re-compile, by using the DOS command (Esc O D).

WARNINGS

ZED cannot cope with running a program which terminates and stays resident (e.g. SIDEKICK or MODE) from within the editor. This DOES NOT mean you cannot run Sidekick with ZED, just that you can't load Sidekick into memory using the ZED (Esc O D) or Alt + D commands, as ZED will then be trapped in memory underneath the memory-resident program.

Under some later versions of Dos, the SET command will not work when run from within the editor. This is because the new copy of COMMAND.COM keeps its own copy of the environment strings, which disappear when COMMAND returns to ZED. Quite why Microsoft should suddenly introduce this unhelpful "feature" without compensating by including something like the Unix "export" command is anybody's guess.

It is quite easy to write programs in C which do horribly illegal things with memory which does not belong to them (e.g. using an uninitialized or NULL pointer), and such a program can easily prevent ZED from regaining control, or otherwise corrupt ZED's memory (and also corrupt DOS). For safety's sake, SAVE ALL of your changed buffers before running another program !

MSDOS Problems

The MSDOS command (Esc O D) and the compile command (Esc O O) will not work on MSDOS versions earlier than 2.11.

Automatic program execution after compile will only work if you are running MSDOS version 3.00 or later. This is because early versions of COMMAND.COM would not allow a command to have a full path name.

If you are running the correct version of MSDOS, and you still have trouble, check you have enough memory. You will need about 256k plus the size of all the files you are editing before you can run even the smallest commands.

If you have enough memory, and have the right version of MSDOS, these commands will still only work if your COMSPEC environment variable is set correctly. This is explained in the MSDOS manual, but is normally automatic unless you boot from a floppy disk (and later remove the boot floppy), or run on a network. In either case, you need to find out where a copy of COMMAND.COM can be found on your fixed disk (e.g. C:\COMMAND.COM), and then add the following line to your AUTOEXEC.BAT file:-

```
SET COMSPEC=C:\COMMAND.COM
```

There should be no spaces in this command line except between SET and COMSPEC.

Record and Playback Menus

You can save a combination of keystrokes in one of 10 macros.

To do this press Ctrl and any function key (F1-F10). Everything you type (up to a maximum of 63 keystrokes) will be remembered against that function key. To remind you that you are recording, the status line shows which numbered macro you are recording (between the cursor column and INSERT/OVERTYPE). To stop the recording, just press Ctrl and the function key again. To recall the macro, just press Alt and the same function key.

You can also record and playback macros using the menus, To record a macro select the record menu (Esc R), and press any of the number keys (0 to 9). To stop recording, select the record menu again, and press the same number key. To recall the macro using the menu method, select the Playback menu (Esc P), and press the appropriate number key.

On the Playback menu, there is an option (Number) to repeat a macro any number of times. If you want to stop the macro repeating once it has started, hold down the Ctrl-Break key. You may save all the macros to a file for recall later by using (Esc R W) to save, or (Esc R R) to read back your macros.

All the macros are remembered, and will automatically be available next time you run ZED.

If you want to save longer macros, you may record keystrokes in a file, by using (Esc R L) to start and stop recording, and (Esc P L) to play back the file. Any playback may be interrupted with a Control - Break.

At any time when you are prompted for a file name, you may enter an ambiguous name (e.g. *.* or \include*.h etc.). A list of all the files satisfying the file name will be shown on the screen. You can then use the cursor keys to move around the list, and press Return to select the file you want. You can also press Escape to take you back to the filename prompt, to enter a different filename.

Text Menu

Update This just re-prints the screen, in case of bugs or other problems. If the text looks funny, or the cursor seems to be in an impossible place, try this. If it still looks wrong, try going to the top of the file with Ctrl + PgUp, and coming back again.

Lmargin (Alt + L) Set the left margin. The left margin is used by the Format command, and during automatic word-wrap. When you press < Return > to end a line, the left margin setting is ignored. This makes it easy to type indented paragraphs, like this one.

Rmargin (Alt + M) Set the right margin - This is where word wrap occurs. If you set it to zero, no word wrap occurs at all.

Tabs (Alt + T) Sets the tab spacing - normally tabs are every 8 characters, but you may set them smaller for heavily-indented programs. The standard DOS PRINT and TYPE commands use a setting of 8, so you should only print source files with a non-standard tab spacing within ZED. If you have such a source file, and you want to print it outside ZED (or edit it with another editor), you can use the Spaces command (below) to convert all tabs to the appropriate number of spaces.

Format (^ B) Used for text files (like this manual) - it sorts out part of the buffer so there is as much text as possible on each line between the left and right margins. The size menu lets you decide how much of the buffer to format, the default being one paragraph (which is defined as the text from the cursor to the next ENTIRELY EMPTY line. Makes an awful mess of most C programs !

Spaces Convert tabs to spaces in part of the buffer. The size menu lets you decide how much of the buffer to format.

tAbs Convert spaces to tabs in part of buffer. The size menu lets you decide how much of the buffer to format.

The quick keys which correspond to these commands are shown in brackets next to each command. These keys may be changed with ZCONFIG. The last three commands may be interrupted with Control-Break.

Notes on formatting and autoindent

If you have the right margin set, word-wrap is automatic between the left and right margins. Pressing carriage return, however, leaves the cursor in column 1, whatever the left margin setting. This is because you normally only press carriage return at the end of a paragraph, and you will want to indent the next paragraph by more or less than the left margin.

If you have auto-indent on, pressing carriage return leaves the cursor in the same column as the first piece of text in the line above. The combination of these two options enable you to set automatic formatting exactly how you like it.

The current margins, and tab settings are saved when you exit the editor, and automatically recalled the next time you invoke ZED.

If you normally use a tab width setting other than the standard 8, and you want to print your files without using ZED, you should convert all the tabs in the whole file to spaces. The Spaces and tAbs commands are also useful if you want to change the tab width setting in a file without changing the appearance of the file - just convert the tabs to spaces using the old tab width, change the tab width to the new setting, and convert the spaces back into tabs.

Backups

Normally ZED keeps a backup of any file you change, with the name as the original, but an extension of .BAK. This is a bit inconvenient if you are editing files with the same name, but different extension (e.g. zed.c and zed.h), as the backups overwrite each other. Also, if you have a hard disk with a complex directory structure, and you run out of disk space, you have to search the whole tree for .BAK files to delete to make space.

If you set the backup directory (using ZCONFIG), ZED will attempt to place backup files on that directory, using the same name as the original. This means you have correctly named backups of all your files, and they are all in one place.

This will only work if the disk containing the original file has a directory with the specified name, as ZED uses the DOS MV system call to move the original file to the backup directory, and this does not work across different disks.

Menu Structure

This is a quick-reference map menu structure. Main menu options are in the left hand column, and sub-menu options across the same line. Underneath each sub-menu option is the quick key, taking you straight to that option.

Main Menu	Sub-menu options											
File	Edit Alt E	Save Alt S	Abort Alt Q	Read Alt I	Write Alt W	eXit Alt X	1-5					
Move	Left Home	toP ^Home	Start ^PgUp	Right End	bottoM ^End	End ^PgDn	Number Alt N	Tag F5	[Alt J	Block Alt B	blocK Alt K	1-9
Edit	Insert (Toggle Insert Key)	Overtyp	Rest F2	Lower	Upper	Setseq	seQuenc Alt U	Comment	+ Alt 0	- Alt 9	Auto Shift+Tab	1-5 Alt 1-5
Blocks	Block Alt Y	blocK Alt Z	Delete	Move	Copy	Save F4	Restore F3	Tag F6	Hilite	Yank	1-9	
Delete	Word F10	End lin F9	Line ^Y	Paragra	Block	Tag	]	All	buf	No of lines		
Search	Find Alt F	Case Alt A	Next F7	Last F8	Again ^L	Replace Alt R	Global Alt G	eXit				
Other	cOmpile Alt C	Compare Alt O	Dos Alt D	Print Alt P	eXit							
Record	1-9,0 ^F1-F10	Write	Read	Log								
Playback	1-9,0 Alt F1-F10	Write	Read	Log	Number							
Text	Lmargin Alt L	Rmargin Alt M	Tablen Alt T	Format ^B	Update	Spaces	tAbs	eXit				
1-5	(Change buffer) Alt 1-5											

Editor commands are also listed in the ZCONFIG section.

Configuration

The program ZCONFIG.EXE is supplied for setting quick keys, directory paths, screen format, screen colors and status line and menu positions.

ZCONFIG creates a file called ZED.CFG containing details of the configuration settings. This file will normally be created on the same directory as ZED.EXE. If ZED.EXE cannot be found somewhere on your PATH, ZED.CFG will be created in the current directory. ZED searches the current directory first, then all the directories on your PATH to find ZED.CFG. Thus if you wish to set up multiple configuration files (presumably to cope with different sorts of editing task), you can copy each ZED.CFG file onto a separate directory, and run ZED in that directory.

The ZED.CFG file is also re-written when you exit the editor, to save the following details :-

ZTC compiler flags

Tab size

Left Margin

Right margin

Block highlighting ON or OFF

Edit mode (Insert/Overtype)

Key macros

Auto-indent ON or OFF

Various options are available once ZCONFIG is run, and these are descibed next.

Set up Quick Command Keys (Hotkeys)

This option allows you to change which key depressions perform which editor commands. A list of all the commands you may perform in the editor is displayed.

Use the cursor keys (Up, Down, PgUp, PgDn, Home, End, Right, Left) to move the cursor to the command you want. Then press the Return key, followed by whichever key (or key combination) you wish to associate with that command. If the key is already in use for another command, you will be asked to confirm that that is the one you want before any changes are made.

If the command already has 3 keys associated with it, the one at the cursor will first be deleted from the list. You can also use the Del key to just delete a key from the list for a command. The commands available are :-

Save file to disk & resume editing	***Esc F S***
Save file and edit new file	***Esc F E***
Abort any changes to this file	***Esc F Q***
Insert contents of a disk file at cursor	***Esc F R***
Write part of the current buffer to disk	***Esc F W***

Save all changed buffers to disk

Leaves you in the editor with all buffers saved and empty

Save file and exit

Saves the current buffer to disk, and clears it. Then steps through the other buffers in numeric order. If a buffer has not been changed, it is cleared. If it has been changed, you are placed in edit mode in that buffer.

Exit program	***Esc F X***
Convert part of the buffer to lower case	***Esc E U***
Convert part of the buffer to upper case	***Esc E L***
Insert ASCII graphic character at cursor	***Esc E G***
Insert sequence number in text, and increment it	***Esc E Q***
Set initial value of sequence number	***Esc E S***
Right-justify 'C' comment on this line	***Esc E C***
Move part of the buffer right 1 tab stop	***Esc E +***
Move part of the buffer left 1 tab stop	***Esc E -***
Toggle auto-indent mode	***Esc E A***
Compile the program in this buffer	***Esc O O***
Compare the contents of two buffers	***Esc O C***
Execute a DOS command	***Esc O D***
Print part of the buffer	***Esc O P***
Search for string	***Esc S F***
Search for a string ignoring case	***Esc S C***
Replace found string with another	***Esc S R***
Replace all found strings	***Esc S G***
Search for a string (Wordstar compatible)	***Ctrl Q F***
Find & replace a string (Wordstar compatible)	***Ctrl Q A***
Save macros to a disk file	***Esc R W***
Read macros back from a disk file	***Esc R R***
Repeat macro many times	***Esc P N***
Set left margin	***Esc T L***
Set right margin	***Esc T R***
Set tab stops	***Esc T T***
Re-format part of the buffer	***Esc T F***
Convert tabs to spaces in part of buffer	***Esc T S***
Convert spaces to tabs in part of buffer	***Esc T A***

Log all key depressions to a file	***Esc R L***
Repeat key depressions from a log file	***Esc P L***
Yank part of buffer into block buffer	***Esc B Y***
Move cursor to a particular line number	***Esc M N***
Tab key	***Tab***
Main menu	***Esc***
Wordstar ^ Q(uick) menu	***Ctrl Q***
Wordstar (bloc) ^ K menu	***Ctrl K***
File menu	***Esc F***
Moves menu	***Esc M***
Edit menu	***Esc E***
Blocks menu	***Esc B***
Delete menu	***Esc D***
Search menu	***Esc S***
Other (Misc) menu	***Esc O***
Recording menu	***Esc R***
Playback menu	***Esc P***
Text formatting menu	***Esc T***
Update screen (in case of bugs)	***Esc T U***
Place text marker 1 at cursor position	***Esc B 1***
Place text marker 2 at cursor position	***Esc B 2***
Place text marker 3 at cursor position	***Esc B 3***
Place text marker 4 at cursor position	***Esc B 4***
Place text marker 5 at cursor position	***Esc B 5***
Place text marker 6 at cursor position	***Esc B 6***
Place text marker 7 at cursor position	***Esc B 7***
Place text marker 8 at cursor position	***Esc B 8***
Place text marker 9 at cursor position	***Esc B 9***

Move to text marker 1	***Esc M 1***
Move to text marker 2	***Esc M 2***
Move to text marker 3	***Esc M 3***
Move to text marker 4	***Esc M 4***
Move to text marker 5	***Esc M 5***
Move to text marker 6	***Esc M 6***
Move to text marker 7	***Esc M 7***
Move to text marker 8	***Esc M 8***
Move to text marker 9	***Esc M 9***
Move to corresponding bracket - (,[,{	***Esc M [***

Mark the start of a block	***Esc B B***
Mark the end of a block	***Esc B K***

Place the TAG at the current cursor position	***Esc B T***
Swap the TAG and the cursor	***Esc M T***
Move to the start of the block	***Esc M B***
Move to the end of the block	***Esc M K***
Show/hide the current block	***Esc B H***
Move marked block to cursor position	***Esc B M***
Copy marked block to cursor position	***Esc B C***
Move text between cursor & TAG to block buffer	***Esc B S***
Copy text between cursor & TAG to block buffer	***Esc B Y***

Enter Insert mode	***Esc E I***
Enter Overtype mode	***Esc E O***
Insert/Overtype mode toggle	***Ins***

Move cursor up 1 line	***Up arrow***
Move cursor to top of screen	***Esc M P***
Move cursor up 1 page	***PgUp***
Move cursor to beginning of file	***Esc M S***
Move cursor down 1 line	***Down arrow***
Move cursor to bottom of screen	***Esc M M***
Move cursor down 1 page	***PgDn***
Move cursor to end of file	***Esc M E***
Move cursor left 1 character	***Left arrow***

Move cursor left 1 word	***Ctrl left***
Move cursor to left of screen	***Esm M L***
Move cursor right 1 character	***Right arrow***
Move cursor right 1 word	***Ctrl right***
Move cursor to end of line	***Esc M R***
Insert newline at cursor position	***Ctrl N***
Scroll text down 1 line	***Ctrl W***
Scroll text up 1 line	***Ctrl Z***
Delete character under cursor	***Del***
Backspace/delete character left of cursor	***Backspace***
Delete word to right of cursor	***Esc D W***
Delete to the end of the line	***Esc D E***
Delete a whole line	***Esc D L***
Delete marked block	***Esc B D***
Search forwards for last pattern	***Esc S N***
Search backwards for last pattern	***Esc S P***
Search for last pattern again	***Esc S A***
Restore last deletion at cursor position	***Esc E R***
Edit buffer number 1	***Esc 1***
Edit buffer number 2	***Esc 2***
Edit buffer number 3	***Esc 3***
Edit buffer number 4	***Esc 4***
Edit buffer number 5	***Esc 5***

Save keystrokes in macro 10	***Esc R 0***
Save keystrokes in macro 1	***Esc R 1***
Save keystrokes in macro 2	***Esc R 2***
Save keystrokes in macro 3	***Esc R 3***
Save keystrokes in macro 4	***Esc R 4***
Save keystrokes in macro 5	***Esc R 5***
Save keystrokes in macro 6	***Esc R 6***
Save keystrokes in macro 7	***Esc R 7***
Save keystrokes in macro 8	***Esc R 8***
Save keystrokes in macro 9	***Esc R 9***
Repeat macro 10	***Esc P 0***
Repeat macro 1	***Esc P 1***
Repeat macro 2	***Esc P 2***
Repeat macro 3	***Esc P 3***
Repeat macro 4	***Esc P 4***
Repeat macro 5	***Esc P 5***
Repeat macro 6	***Esc P 6***
Repeat macro 7	***Esc P 7***
Repeat macro 8	***Esc P 8***
Repeat macro 9	***Esc P 9***
Help	***F1***

Path Settings (Files)

Ramdisk Where to put temporary files during compilation. MUST include the colon (disk name) or terminating backslash (directory). (eg \TMP\ or D:)

Help The full pathname of the ZED help file (eg C:\ZORTECH\BIN\ZED.HLP)

Backup Directory on which to put backup files. This MUST NOT include a disk name, as the backup files are always held on tha same disk as the original. See BACKUPS above.

Libs The path where you keep the Zortech libraries (*.LIB & *.OBJ). MUST include the colon (if a disk name) or terminating backslash (if a directory). (eg C:\ZORTECH\LIB\ )

Include The path where you keep the Zortech include files (*.h & *.hpp). MUST include the colon (if a disk name) or terminating backslash (if a directory). (eg C:\ZORTECH\INCLUDE\ )

Printer Name of printer device (e.g. PRN or LPT1 or COM1)

Eof Include ^ Z character at end of files. This switch needs to be ON for compatibility with Wordstar. Some other programs do not recognize ^ Z as an end-of-file marker, and need this switch to be OFF.

Screen Display (Screen)

Scroll

Use hardware scroll for scrolls up to < n > lines : default 10. ZED will scroll the text on the screen if it is moving up to this number of lines, and will just reprint the whole screen if it is moving more than this number. Some screens give an annoying flicker when scrolling more than a few lines. If your screen does this, set this number to a small value. If your scroll is smooth, fast, and flicker-free, set this number to a high value (up to the maximum number of lines on your screen).

Lines

Scroll n lines when PgUp or PgDn keys pressed : default 10. If you prefer PgUp and PgDn to move a whole screenful at a time, set this equal to the number of text lines ZED displays on your screen (22 for most screens).

Update

Update at least n lines when scrolling : default 1. If a key is pressed before the screen is up to date, the screen update is stopped, but not before at least this number of lines have been updated. The larger this number, the more up to date the screen is kept when you are scrolling or paging quickly through the file, but the the paging then occurs more slowly.

Horizon

Scroll n columns when cursor goes off of screen : default 16. ZED normally does sideways scroll in jumps of 16 characters at a time. If you prefer larger or smaller jumps, change this value.

Tab Character

ASCII char to represents tabs : default 26. 26 is the little right arrow used to represent tabs in the ZED display. If you don't like tabs to be shown, set this value to 32 (i.e. a space)

Newline

Use ASCII char n to represent end of line : default 27. Character 27 is the little left arrow used to represent end of line in the ZED display. If you don't like line ends to be shown, set this value to 32 (i.e. a space)

Eof

Use ASCII char n to represent end of file : default 186. Character 186 is the double vertical bar used to show that the screen display has gone past end of file. If you don't like this to be shown, set this value to 32 (i.e. a space)

Ruler

Use ASCII char n to fill the ruler line : default 205. Character 205 is a double horizontal bar.

Video

Set video mode : default 3 (80x25 color). This is the parameter to the "set video mode" BIOS call (INT 10 hex, AH = 0, AL = this value). Set to these values:-

- **2** Color Graphics adapter 80x25 mono
- **3** Color Graphics adapter 80x25 color
- **7** Mono Display adapter 80x25 (if you don't have a color screen)
- **55** ATI EGA Wonder card 132x44 mono
- **99** Enhanced Graphics adapter 80x44 color. This is a special value supported by ZED)

ZED does NOT support modes with less than 80 columns. If you know of other valid video modes that work with your machine, you may use them. With certain BIOS/monitor combinations it is possible to damage your monitor by setting the video mode to an illegal value, so BE VERY CAREFUL.

Cursor Set main edit cursor shape : default 000F (Hex). This is the parameter to the "set cursor shape" BIOS call (INT 10 hex, AH = 1, CX = this value). This value must be entered in Hex. The high byte (1st two hex characters) is the start of the cursor block, and the low byte (next two hex characters) is the end of the cursor block. On most monitors 000F gives a block cursor, 0B0C gives an underline cursor.

Moves Toggle whether cursor right at the end of a line moves to the next line and whether cursor left at the start of a line moves to the previous line. By default, if you hold down the cursor left (or right) keys, the cursor stops at the beginning (or end) of the line. If you switch this option, the cursor will jump to the end of the previous line (start of the next line), just like Wordstar. This option toggles the switch on and off.

Dumb Toggle intelligent cursor movements on/off. Cursor movement is normally intelligent, i.e. the cursor will return to the same position if you move it up or down and then back to the same line, even if it had to be in a different column position on another line. If you switch this option, the cursor will act like Wordstar. This option toggles the switch on and off.

Screen colors (colors and More colors)

menuBar	Color for menu line .
menuOption	Color for highlighted menu option.
menuPrompt	Color for menu prompt.
Strings	Color for input strings etc.
Warnings	Color for editor warning and error messages.
Window	Color for window menus.
Option	Color for highlighted option in window menus.
Status	Color for status line.
Ruler	Color for ruler line.
Errors	Color for compiler error messages .
Detail	Color for compiler error details.
Text	Color for text being edited.
Block	Color for highlighted block.
Help	Color for help text.

The position of the cursor on the top line is shown by highlighting the character under the cursor.

Overtype	Color for overtype cursor on input line (normally black on white)
Insert	color for insert cursor on input line (flashing inverse)

Move the cursor to the 'Z' which is the color you want, and press return. If you decide not to change the color, press Esc.

Screen Layout (Lines)

Allows you to change the screen layout. Select a line number between 1 and 8 for each item. You may overlap these lines as required. Some overlaps may give a strange display.

Menu	The line for the menu bar (default 1)
Prompt *	The line for the menu prompt/explanation (default 2)
Status	The line containing status information (default 2)
Warnings	Editor error messages (default 1)
Input	The line for user input of strings etc. (default 1)
Ruler *	Where to put the ruler line (default 3)
Errors	The line for compiler error messages (default 2)
Detail	The line for compiler error message details (default 3)

* The Prompt and Ruler lines may be turned off altogether by setting their line numbers to 0.

Window Positions

The window menu allows you to change the positions of the menu windows. Select which menu you wish to position, and use the cursor keys to move it around the screen to the required position. As you can see from the defaults, you may overlap these menus as required. Some overlaps may give a strange display, particularly where menus overlap status line displays.

You can move any menu window so that part of it disappears off the bottom of the screen. This will disable some of the later options on that menu, as the cursor would need to move off the bottom of the screen to select them !

If you prefer your menus on the top line (Lotus 123 style), move each menu window to the top left hand corner of the screen The menus available for selection are :-

Main	Main menu
File	File menu
Move	Move menu
Edit	Edit menu
Blocks	Blocks menu
Delete	Size menu
Search	Search menu
Other	Other menu
Record	Record menu
Playbac	Playback menu
Text	Text menu
K	Wordstar ^ K (block) menu
Q	Wordstar ^ Q (quick movements) menu

Boxes

The box menu allows you to change the characters used for the borders of the window menus. Just enter the ASCII character number for each position. The default characters are the single line box characters :-

menu option	part of border	default value
7	top left corner	218
8	top edge	196
9	top right corner	191
4	left edge	179
6	right edge	179
1	bottom left corner	192
2	bottom edge	196
3	bottom right corner	217

Details of the ASCII value of all the IBM graphics characters can be found in your DOS manual, or you can experiment using the (Esc E G) function in ZED.

Printing Configuration (Print)

You can print the current configuration settings (including the "hotkeys"). The print goes to the printer device as set in the Directories/Print menu.

Saving Configuration (saVe) and Exiting (eXit)

Help File

The standard ZED help file is called ZED.HLP. If you change the hotkeys using ZCONFIG, you may wish to change the help file accordingly.

As the help file is a standard text file, it may be edited using ZED like any other file. Please note that you should not edit the help file while ZED is actually using it ! Please copy the file ZED.HLP to another filename, edit the new filename, and then copy it back again.

The help file consists of a number of pages of help information. Pages are separated by form feed characters (Graphics character 12). When you edit the file, make sure you don't make any pages too big to fit on the screen. The pages are numbered (the first page is number 0) for reference when creating help cross-reference options.

Help cross-reference options are indicated by a right-pointing triangle character (Graphics character 16), the page number the option will go to, the highlighted option text, and a left-pointing triangle character (Graphics character 17). At the very least, you should include a help cross-reference option pointing to the help index page (page 20) on every help page. Up to 40 pages of help are allowed, and up to 50 cross-reference options on each page.

ZORTECH COMPILER

Overview of the Compilation Process

1. C++ source files are created by the user using a standard ASCII text editor. A word processor in unformatted / non- document mode may be used.

2. ZTCPP1 is run to process the source file and to create a temporary intermediate file.

3. ZTC2 is run on the output of ZTCPP1 to create a standard Intel .OBJ file.

4. Steps 1, 2, and 3 are performed for each file in the C++ program.

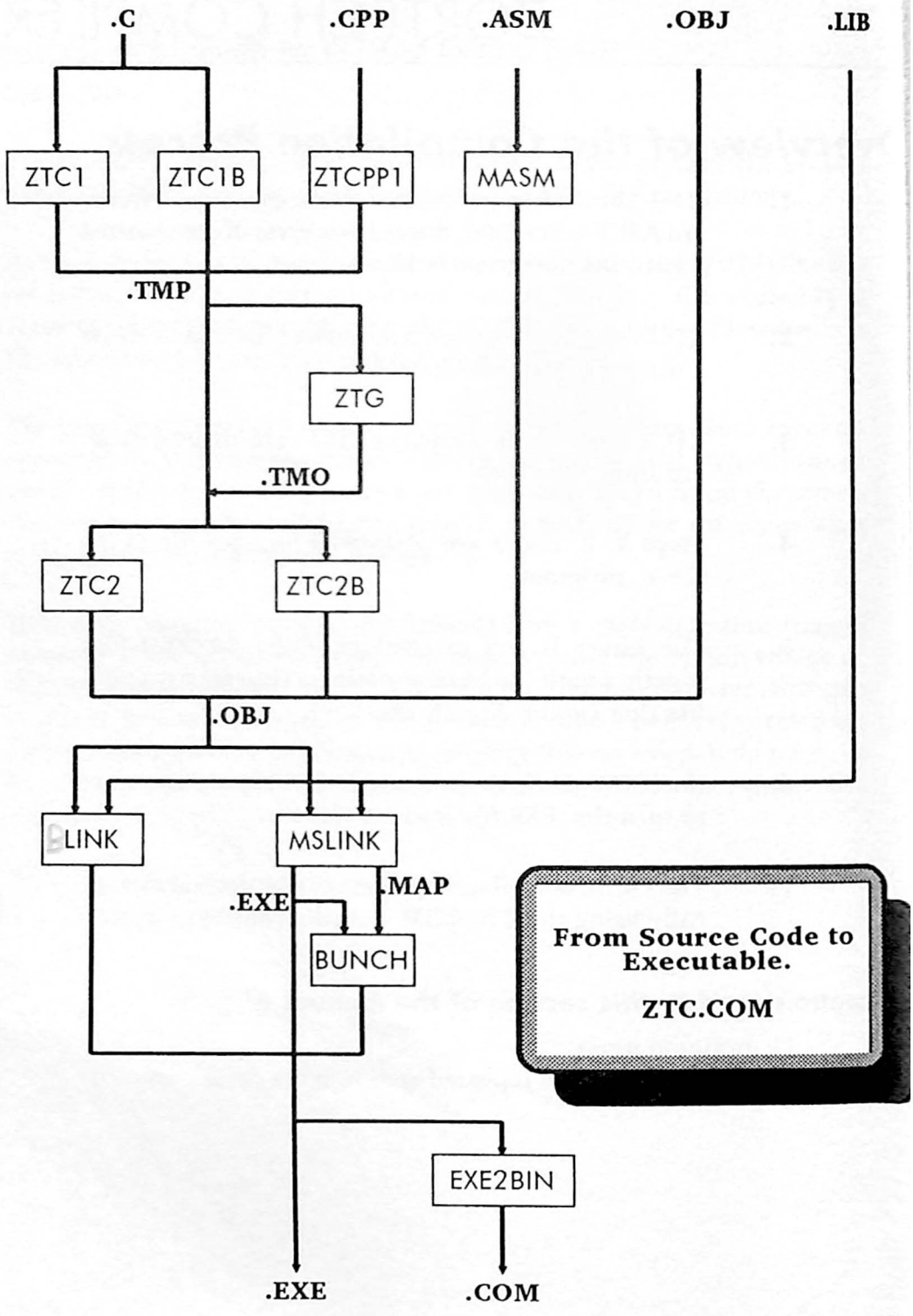
5. The linker is run to combine all of the .OBJ files together with the library routines. The result is a .EXE file that can be directly executed.

6. If a .COM file is desired, the EXE2BIN program is run to turn the .EXE file into a .COM file.

7. The entire compilation process can be done automatically using the ZTC.COM compiler control program.

Notation used in this section of the manual

[] optional items.
{} items that can be repeated zero or more times.


.C
.CPP
.ASM
.OBJ
.LIB
ZTC1
ZTC1B
ZTCPP1
MASM
.TMP
ZTG
.TMO
ZTC2
ZTC2B
.OBJ
LINK
MSLINK
.MAP
.EXE
BUNCH
From Source Code to Executable.
ZTC.COM
EXE2BIN
.EXE
.COM

ZTC1.EXE, ZTC1B.EXE and ZTCPP1.EXE

All remarks about ZTC1.EXE also apply to ZTC1B.EXE and ZTCPP1.

ZTC1.EXE and ZTCPP1.EXE are the first pass of the C and C++ compilers, respectively. They read the source file, execute all preprocessor functions, check for all syntax errors, and generate a compressed intermediate form of the file suitable for input to ZTG or ZTC2.

ZTC1B is simply a version of ZTC1 that can handle much more complex programs without running out of memory, though it is much slower.

Command Line

Command line switches and parameters are:

To compile a C source file:

```
ZTC1 infile {switches}
```

To compile a C++ source file:

```
ZTCPP1 infile {switches}
```

infile

This is the input source file. If the extension is omitted, it is assumed to be ".C" for ZTC1 and ".CPP" for ZTCPP1.

switches

=nnnn Sets the size of the compiler's internal stack. Try this if the compiler exits with a message about 'stack overflow'. Good values for nnnn to try are (in decimal):

ZTC1	2000 to 10000
ZTC1B	5000 to 30000
ZTCPP1	5000 to 30000

-a Suppress alignment within structures. Members within structs are normally aligned on 16 bit boundaries to maximize speed on machines with a 16 bit bus (like the AT).

-A Enforce strict ANSI compatibility.

Causes the equivalent of

#define __STDC__ 1

to be executed. Also, the following keywords will no longer be recognized:

far huge near pascal cdecl

-co Link for CodeView. Must use Microsoft LINK.

-d[macro[=text]]
-D[macro[=text]]

If the switch '-d' is specified it has the effect of:

#define DEBUG 1

appearing before the compilation starts. The switch '-dmacro' has the effect of

#define macro 1

The switch '-dmacro=text' has the effect of

#define macro text

There can be no embedded white space. If embedded white space is desired, enclose the entire switch in " " or ' ', as in '-Dmacro="this is a string"' which is equivalent to

#define macro "this is a string"

-e Show the effects of the preprocessor in both error messages and the list file, if specified. This is good for verifying if macros are expanding as expected.

-f Generate in-line 8087 or 80287 instructions. This will significantly speed up the code, reduce its size, and improve its accuracy. See the section on floating point options.

-g Insert line number information in the object file for debugger support. This is compatible with Microsoft's SYMDEB and CODEVIEW debuggers. Line number information will significantly increase the size of the .OBJ file.

-iincludepath
-Iincludepath

Search for #include files along this path. Up to ten include paths may be given. Directories are searched for #include files in the following order:

For < . . . > files:

-I paths, in the order specified.
INCLUDE environment variable paths.

For " . . . " files:

The default directory. -I paths in the order specified.
INCLUDE environment variable paths

-I Paths, in the order specified.

INCLUDE Environment variable paths.

-j[0|1|2] Recognize two byte Asian language characters within character constants and strings. That is, if a character matches the start of a two byte sequence, the following byte is not checked to see if it is a \ or the closing quote. The second byte cannot be a 0, carriage return (0x0D) or end-of-file (0x1A).

0 Japanese (default)
0x81..0x9F and 0xE0..0xFC signal the start of a two byte sequence.

1 Taiwanese and Chinese
0x81..0xFC signal the start of a two byte sequence.

2 Korean
0x81..0xFD signal the start of a two byte sequence.

-J Treat chars as being unsigned by default. This makes the type for <char> be <unsigned char>, and "string" becomes <unsigned char *>. This switch is useful for quick ports of code that depends on chars being unsigned (signed is the default). Note that the behavior of the runtime library routines are not affected by this switch, unless they are recompiled also.

Dependence on whether chars are signed or unsigned is non-portable and will cause problems sooner or later. The recommended practice is:

signed char	Code depends on it being signed.
unsigned char	Code depends on it being unsigned.
char	Code doesn't depend on whether it is signed or not.

If later on the code needs to be ported to an older compiler that doesn't support the signed keyword, do this:

```
#ifndef __ZTC__          /* if not using Zortech C */
#define signed           /* eliminate the keyword */
#endif
```

-l[listfile] Generate a listing file with the name listfile. If listfile is omitted, then the name defaults to that of the input file with the extension ".LST". The effects of the preprocessor can be shown in this file if the -e switch is also specified. Error messages and line numbers are inserted into the file.

-m[S|M|C|L]

The memory model is selected using the -m switch followed by a letter specifying the desired memory model. They are as f ollows (with maximum sizes):

Model	Code	Data
S - Small	64k	64K
M - Medium	1 meg	64k
C - Compact	64k	1 meg
L - Large	1 meg	1 meg

No, the T memory model is not forgotten. A T model program (.COM) is created by linking an S model program a special way. See the section on creating .COM files. For more information see the *Advanced Features* and *Interfacing with Assembler* sections.

-ooutfile The output file name is normally the same as the input file name with the extension of ".TMP". With the -o switch, a different file name may be specified.

-p Turn off autoprototyping. This is provided so that old code can be compiled without worrying about it. See the section on autoprototyping in Part Two.

-r Enforces strict prototyping. Functions must be declared and prototyped before use. Note that int func() will be interpreted as int func(void).

-u Suppresses the definition of the predefined macros. See the section on predefined macros in Part Two.

-v Verbose compile. Displays each function name as it is encountered. Useful for determining where in the source file the compiler was if trouble occurs.

-w Suppress warning error messages.

-x Does not stop the compiler when the error limit is reached. The compiler continues compiling and displays all errors detected.

Environment Variables

The environment variable INCLUDE is examined to obtain the search path for #include files. The format is the same as for the DOS PATH environment variable. Example (of a command that could be placed in AUTOEXEC.BAT):

```
SET INCLUDE=C:\ZORTECH;C:\ZORTECH\INCLUDE
```

This would cause ZTC1 or ZTCPP1 to add the directories C:\ZORTECH and C:\ZORTECH\INCLUDE to the search path for #include files.

Examples

1. *ZTC1 hello*
   Compile the input file hello.c and create the output file hello.tmp.

2. *ZTC1 hello -od:xyz -l -dabc i\compiler*
   Compile the input file hello.c and create the output file d:xyz. Create a listing file named hello.lst. Define abc. Search the path \compiler\ for any #include files.

ZTG.EXE

ZTG.EXE is the global optimizer program. It is run on the output file produced by ZTCPP1.EXE, ZTC1.EXE or ZTC1B.EXE and its output file becomes the input for ZTC2.EXE or ZTC2B.EXE.

No syntax errors occur during this pass. ZTC1 detects all errors in the source file. The input file is deleted on successful completion of ZTG.

Command Line

ZTG input [-ooutput] {(+|-)switches} [-v] [=nnnn]

input	File that is the output from ZTCPP1, ZTC1, or ZTC1. An extension of .TMP is assumed if one is not given.
output	Output file name for ZTG. If not specified, the output file name is formed by taking the input file name and appending the extension .TMO to it.
-v	Verbose compile (list function names as they are processed).
=nnnn	Set ZTG's internal stack size. Try this if you get a "Stack Overflow" message from ZTG. Good values to try are 5000 . . . 3000.

Other Switches

For the rest of the flags, + means turn on optimization, - means turn it off.

all	Perform all optimizations (default).
cnp	Constant propagation.
cp	Copy propagation.
da	Dead assignment elimination.
dc	Dead code elimination.
dv	Dead variable elimination. Compute live ranges.
gcse	Global common subexpressions.
li	Loop invariant removal.
liv	Loop induction variables.
loop	Iterate until no more optimizations.
none	Do no optimizations.
reg	Do our own register assignments.
space	Favor space optimizations over execution time optimizations.
time	Favor execution time optimizations over space optimizations (+ time is the default).
vbe	Very busy expressions.
w	Turn on warnings (like variable used before it's defined).

Examples:

```
ZTCPP1 demo
ZTG demo
ZTC2 demo
LINK demo;
```

means compile, optimize, and link the program demo.cpp.

```
ZTG demo +space -w
```

means favor code space optimizations, turn off warnings.

ZTC2.EXE and ZTC2B.EXE

All remarks about ZTC2.EXE also apply to ZTC2B.EXE. ZTC2B is a version of ZTC2 that can handle more complex programs without running out of memory, though it is much slower.

ZTC2 infile [-ooutfile] [-s] [-v] [=nnnn]

ZTC2 is the 8086 code generator that takes the output file created by ZTCPP1, ZTC1, ZTC1B, or ZTG and turns it into a .OBJ file suitable for linking. No syntax errors occur during this pass. ZTC1 detects all errors in the source file. The infile is deleted on successful completion of ZTC2.

infile

This is the input source file. If the extension is omitted or is .C, it is assumed to be .TMP or .TMO (.TMO files are output by ZTG).

switches

=nnnn Sets the size of the compiler's internal stack. Try this if the compiler exits with a message about 'stack overflow'. Good values for nnnn to try are (in decimal):

ZTC2	2000 to 10000
ZTC2B	5000 to 30000

-ooutfile The output file name is normally the same as the input file name with the extension of .OBJ. With the -o switch, a different file name may be specified.

-s Causes stack overflow checking to be inserted at the beginning of each function.

-v Run compiler in verbose mode. Prints out the name of each function as it is processed.

ZTC Command

ZTC is a compiler control program. The ZTC command is similar to the cc command in UNIX. ZTC runs all passes of the compiler, the linker, BUNCH, EXE2BIN, and even MASM if requested. A single or multiple file program can be created with this single command.

ZTC can handle different types of files including C source, C++ source, assembly source, object, and object libraries. C source files (.C) are compiled with ZTC1. C++ source files (.CPP) are compiled with ZTCPP1. Assembly files (.ASM) files are assembled using MASM.

Object files (.OBJ) are included in the link step. Library files (.LIB) are included in the link step in the libraries section.

The syntax for ZTC is -

```
ZTC {switches} file1 {file2}
```

The output file (unless the -ofilename overrides it) is the first filename with the path and extension stripped and the appropriate extension (.OBJ, .EXE or .COM) appended. ZTC does wildcard expansion on filenames. If files are given without extensions, ZTC tries extensions in the following order in an attempt to find the file:

.CPP .C .ASM .LIB

The switches for ZTC include all switches for ZTC1 and ZTC2 plus :

-A Enforce strict ANSI compaibility.

-b Use ZTC1B and ZTC2B, useful if ZTC1 or ZTC2 runs out of memory.

-B Run BUNCH on resulting **.EXE** file. This is necessary if one of the modules is a C++ module and Microsoft LINK is used.

-c Compile/assemble the source files only. Do not link them together.

-co Link for CodeView. MS-LINK must be used for this, as Zortech LINK does not support CodeView.

-cpp Use ZTCPP1 for both **.C** and **.CPP** files. Normally ZTC will attempt to use ZTC1 or ZTC1B for files with an extension of **.C.**

-g Cause a **.MAP** file to be generated by the linker. Causes debug information to be put into **.OBJ** files.

-m[M][i] Define memory model and if program is to be integer only. The **i** determines if it is integer only (it causes INT.OBJ to be linked in). **M** is one ofthe following: CT.OBJ

S	S model (default)
M	M model
C	C model
L	L model
T	T model (see **.COM** programs)

Examples:

without -m
Small model **.EXE** with floating point.

-msi	Small model integer only.
-mt	*COM* file with floating point.
-mti	**COM** file integer only.

-o Run optimizer (ZTG.EXE).

-o+flag
-o-flag Run optimizer passing it +flag or -flag.

-ofilename
Specify output file. This is the **.OBJ** file if -c is also given, otherwise it is the **.COM** or **.EXE** file.

=nnnn Set stack size for ZTCPP1, ZTC1, ZTC1B, ZTG, ZTC2 and ZTC2B where **nnnn** = size in bytes. The commands to the programs are displayed as they are executed. If the command line to LINK would exceed 127 characters, ZTC generates a linker response file with the name exefile.LNK and executes the linker with that.

-r Enforce strict prototyping. All functions must be declared and prototyped before use. Note that if this flag is used int func() will be interpreted as int func(void);

-v Run the compiler in verbose mode (this will also cause the version number to be displayed).

Environment Variables

ZTC uses the following environment variables:

INCLUDE Actually used by ZTC1, ZTC1B and ZTCPP1, this specifies the path to search for #include files. This path is searched after any paths specified by -I.

TMP Path to where the intermediate files are written. If not specified, it defaults to the current directory. This is also used by LINK.

LIB Actually used by LINK, this specifies the search path for libraries.

CFLAGS Expands to string that acts as if it was typed on the ZTC command line. For example, to turn on verbose compiles and always use ZTC1B and ZTC2B:

SET CFLAGS=-v -b

Examples

1.

```
ZTC -c myprog -o+space
```

Compile myprog.c into myprog.obj, optimizing for space.

2.

```
ZTC myprog
```

Compile and link myprog.c into myprog.exe

3.

```
ZTC -ml file1.c file2.c
```

Compile file1 and file2 using large memory model then link into FILE1.EXE.

4.

```
ZTC myprog.c mod.cpp myprog1.asm my.lib myprog2.obj
```

a. C compiles MYPROG.C.
b. C++ compiles MOD.CPP.
c. MASM's MYPROG1.ASM.
d. Links MYPROG.OBJ, MOD.OBJ, MYPROG1.OBJ, MYPROG2.OBJ, and MY.LIB into MYPROG.EXE.

5.

```
ZTC *.c -otest.exe
```

Compile all .c files in this directory, and link them to form TEST.EXE.

C++ REFERENCE

Comments

In C++, // starts a comment, which terminates at the end of the line on which they occur. C++ also supports the C commenting convention using the symbols /* and */ to start and end a comment. These may not be nested. To comment out blocks of code, it is suggested that #if 0 and #endif be used.

Identifiers

Up to 127 significant characters are allowed in an identifier. More than that is flagged as a syntax error. Underscores, letters, and digits are allowed, and case is significant.

Keywords

The following words are reserved :

auto, break, case, char, class, const, continue, default, delete, do, double, else, entry, enum, extern, far, float, for, friend, goto, if, inline, int, long, near, new, overload, operator, public, register, return, short, signed, sizeof, static, struct, switch, this, typedef, union, unsigned, virtual, void, volatile, while

Identifiers that should be avoided because of conflicts with other implementations are:

asm, cdecl, fortran, huge, pascal

Conflicts With Identifiers in the Library

Care must be taken in using external identifiers that begin with '_' , as they may conflict with library routines.

C++ makes heavy use of overloaded meanings for names, so there must be many global symbols created that are spelled one way within C++, and another way for the linker, ie. C++ must generate many of the global symbols. To maximize compatibility with debuggers, the generated names are in the same name space as normal C identifiers. There are potential name conflicts. To minimize this possibility:

- *Never name identifiers with leading or trailing _.*

- *When using global overloaded function names, declare them in the same order in each module. Actually, only the first one must be declared first in each module. That is because only the first one does not undergo any name transformation, because otherwise you would not be able to overload names that appear in the library.*

- *Examine the linker .MAP file a few times to get a feel for the types of names that the compiler generates, and avoid using similar names.*

- *If you use constructors or destructors for static variables, don't use the same source file to generate more than one object file which will all be linked together. The reason is that the name generated for static constructor and destructor functions depends on the source filename for uniqueness. For example, the file test.cpp will generate the global names:*

  _STItest_cpp_ and _STDtest_cpp_

Character Constants

The standard character constant is enclosed in single quotes 'a'. Characters may also be represented as escape sequences. The escape sequence represents the bit pattern value of the desired character in either hex or octal format. An escape sequence begins with the \ backslash.

bit pattern	\nnn	octal value of desired character
bit pattern	\xnn	hex value of desired character

The escape sequence \0 is a special case representing the character NULL. Other characters may be represented by the following escape sequences:

new-line	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
backslash	\\
single quote	\'
double quote	\"

Multiple character constants are allowed. For example, 'ab' is equivalent to 0x6162. 'abcd' is equivalent to 0x61626364L.

Types

The following basic data types are implemented as:

char	=> signed 8 bits
unsigned char	=> unsigned 8 bits
int	=> signed 16 bits
unsigned	=> unsigned 16 bits
long	=> signed 32 bits
unsigned long	=> unsigned 32 bits
float	=> 32 bit floating
double	=> 64 bit floating
near pointer	=> 16 bit segment offset
far pointer	=> 16 bit segment and 16 bit offset

Values are stored in memory with the least significant byte at the lowest address. The following basic data types are equivalent:

short	=> int
signed short	=> int
unsigned short	=> unsigned
unsigned int	=> unsigned
signed int	=> int
signed long	=> long
long float	=> double
long double	=> double

const and volatile

The const and volatile type specifiers define how and when an object will be accessed by the compiler. The const type specifier declares an object to be a constant that cannot be assigned to. An object may be declared const because the object is in ROM or because the object is not to be modified (e.g. a data table). The compiler will flag assignments to const objects as an error. The compiler can generate more compact code when it knows an object is constant. The volatile type specifier is used to tell the compiler that an object may change in value even if the program does not explicitly change it. An object should be declared volatile if it is a memory mapped device register or a data object shared by separate processes.

The volatile type solves a problem for programmers who work with volatile data using an optimizing compiler. Compilers with minimal optimization read the variable each time its value is needed. Compilers performing greater optimization, like Zortech C++ , read the value of a variable the first time the variable is needed and then save that value in internal registers for successive reads of the variable. When an object is declared volatile, then an optimizing compiler reads the variable each time it is required.

Example

```
const int a=3;  /* a is guaranteed to never change */
volatile int flag;
                /* flag could be set asynchronously */
func()
{  flag = 0;
   while (flag==0)
            ;                    /* wait for interrupt */
}      /* presume that service() is called whenever a
                                timer interrupt occurs */
service()
{
   flag =1;     /* indicate that interrupt occurred */
}
```

extern and Global Variables

There are three types of global declarations:

```
extern int v1;
int v2 = 67;
int v3;
```

The first declaration means that storage for v1 is actually allocated in some other module. The second means that storage for v2 is allocated in this module, and v2 can be referenced by extern declarations in other modules. The third one creates a common block for v3.

Common blocks are created when a variable is declared as being global but no initializer is provided. What it means is that the common block declaration can appear in many source modules, but only one instance of v3 will actually be allocated, and all the declarations of v3 will refer to that one.

If there is a common block declaration for a variable in one module, and an initialized global declaration for it in another module, the initialized global declaration will allocate the storage for the variable and the common block declarations will refer to it.

It is an error to declare initialized global data with the same name in multiple modules. It is also an error to declare extern variables but nowhere have a common block or initialized global data declaration for it. Common blocks are allocated in the c_common segment, and are initialized to zero by the startup code in C.OBJ. Common blocks appear in the .OBJ file as COMDEF records, which are a Microsoft-defined extension to the standard Intel .OBJ file format.

Alignment

All data objects are aligned on 16-bit word boundaries. This makes code executing on CPU's with a 16-bit data bus run faster (like the IBM-AT). Alignment can be disabled within structures by using the "-a" switch during compilation. In the case of two or more repeated single character variables occurring in the structure, the first will be aligned on a word boundary, whereas subsequent characters will be byte aligned.

Function Parameters

Some types given as function parameters are modified:

char	is changed to	int
float	is changed to	double
array of	is changed to	pointer to
unsigned char	is changed to	unsigned

Examples

```
int func(a,b,c,d)
char a;
short b;
float c;
int d[];
```

is equivalent to:

```
int func(a,b,c,d)
int a;
int b;
double c;
int *d;
```

Pointers

Pointers are type checked. A cast may be used to escape from the typing rules. For example:

```
char *p;
int *a;
a = (int *) p; /* use cast to prevent syntax error */
```

Initializers

Statics and globals are initialized to 0 unless otherwise specified. Autos are garbage unless explicitly initialized. Unions can be initialized using the first member rule.

Example

```
union a
{
   int i1;
   float f1;
   double d1;
} union_a = 0x123;          /* initializer for i1 */
```

Floating Point Operations

All floating point operations are done as doubles, e.g:

```
float a, b, c;
a = b * c;
```

is equivalent to:

```
float a, b, c;
a = (float) ((double) b * (double) c);
```

Note that both +0 and -0 are representable, and both mean 0.

Floating point errors can be intercepted by rewriting the CXFERR routine provided in the library sources (under FLOAT.ASM).

The Preprocessor

The compiler's first pass contains an integral preprocessor. This supports all directives defined in the ANSI draft. Preprocessor directives begin with '#' as the first non-blank character on a line. Spaces and tabs can appear between the '#' and the directive. Supported preprocessor directives are:

#if, #else, #endif, #elif, #ifdef, #ifndef, #error, #line, #define, #undef, #include, #pragma, #message, and #exit. #pragma is ignored.

The effects of the preprocessor can be viewed by using the '-e' and '-l' command line switches to ZTC1 or ZTCPP1.

Macros

Full macro capabilities are supported. Circular macro definitions such as

#define unix unix

expand to 'unix'. Argument substitution does not take place inside strings. For example, if the preprocessor encounters the following definition :

```
#define X(y) printf("y = %d\n", y)
```

then the following :

```
X(abc);
```

will expand to :

```
printf("y = %d\n", abc) ;
```

Note that the y in the string was not replaced. If the macro is written as :

```
#define X(y) printf(#y " =%d\n",y);
```

it will expand to :

```
printf("abc" " =%d\n",abc);
```

The adjacent strings will be concatenated.

Predefined Macros

The following macros are predefined by the compiler. They may be disabled with the -u switch.

__ZTC__	Expands to the version number of the compiler (in hex). For example, version 4.02 would expand to 0x402.
DLC	Expands to the same thing as __ZTC__. This is here for backwards compatibility with Datalight C. Do not use it for future development.
MSDOS	Expands to 1.
I8086	Expands to 1.

One of the following is defined and set to 1 depending on the memory model:

M_I86SM
M_I86MM
M_I86CM
M_I86LM

SPTR	Defined / set to 1 for S and M memory models.
LPTR	Defined / set to 1 for C and L memory models.
c_plusplus	Defined and set to 1 if ZTCPP1 is used.

The following follow ANSI C guidelines and cannot be disabled with the -u switch:

__LINE__	Set to current line number in the source file.
__FILE__	Set to a string which is the current source file name.
__DATE__	Set to a string which is the current date.
__TIME__	Set to a string which is the current time.

The following is defined and is set to 1 if the -A switch is set:

__STDC__

Include Files

```
#include "filename"
```

Filename is searched for in the default directory. If it is not there then the list of paths provided with the '-I' command line switch is searched for the filename, then the path provided with the INCLUDE environment variable is searched.

```
#include <filename>
```

Filename is searched for along the list of paths provided with the '-I' command line switch followed by the path specified with the INCLUDE environment variable. If no '-I' path, or environment variable INCLUDE path was specified, then the default directory is searched for filename.

The backslash character in include path strings is not used as an escape character. This allows the include path to be written in the include statement the same as on the DOS command line or elsewhere. Some MS-DOS compilers require the string to have two backslashes(\\) to separate directory names.

For example:

```
#include "c:\include\stat.h"            // ZTC now */
#include "c:\\include\\stat.h"   // older compilers */
```

Other Preprocessor Directives

Two non-standard preprocessor directives are supported: #message and #exit

1. #message string new-line

The string is written to the display during compilation.

Example

```
#if INTONLY
   #message "compiling for integer only"
#endif
```

2. #exit constant-expression new-line

This causes the compiler to halt the compilation, close all the output files, and return with an exit status given by the constant-expression. The exit status can be tested in a batch file using the DOS ERRORLEVEL function

Example

```
#if sizeof(int) != 2
#message "ints must be 2 bytes wide"
#exit 1      /* abort compilation with errorlevel 1 */
#endif
```

Limitations

Characters in a line:	*no limit*
Characters in an identifier:	*127*
Characters in an external identifier:	*127*
Characters in a string:	*no limit*
Number of cases in a switch:	*no limit*
Characters in an argument to a macro:	*no limit*
Number of arguments to a macro:	*127*
Number of arguments to a function:	*no limit*
Length of macro replacement text:	*no limit*
Number of subscripts in an array:	*no limit*
Complexity of a declaration:	*no limit*
Number of #includes that can be nested:	*14*
Number of #include paths:	*10*
Number of #if's that can be nested:	*10*
Number of command line arguments:	*32*

with main x.obj unlimited.

NOTE: no limit means that there is no compiler limit. Memory constraints apply.

Function Prototyping

Prototyping allows you to declare the number and type of arguments passed to a function. From then on the compiler checks each function call and flags an error for the calls that do not match the prototype declaration. This type checking feature performs much of the function of the UNIX LINT utility.

Prototype declarations can have several different forms for the various ways C++ functions are called. The prototype declaration can specify each argument, specify some arguments, let there be an arbitrary number of arguments with any type, or declare that no arguments can be passed. Also, a new type used with prototypes is the generic pointer.

Prototype function declarations are much the same as the old "extern func(); " declarations, but with the addition of argument type descriptions. These argument type descriptions are an abstract version of the type. The actual name of the argument is not included in this declaration.

For example -

```
extern int fseek(FILE *, long, int);
```

This function declaration declares fseek to be a function with three arguments. The first argument is a pointer to type FILE, the second is a long integer, and the third is an integer. The declaration also defines fseek to return an integer. A function that will never have any arguments is declared with a "void" argument list.

```
abort(void);
```

This function declaration states that abort should never be called with any arguments.

Function declarations with an unknown number of arguments of unknown type are formed using the old external function declaration. This provides compatibility with older programs.

For example :

```
extern int printf();
```

This is a completely legal declaration although it can be improved because the first argument to printf is known to be a character string. A better declaration for printf would be:

```
extern int printf(char *, ... );
```

Now the first argument to printf must be a character string, but all remaining arguments may be of any number and type.

A "generic pointer type" is used when the type of pointer may vary based on the application. For example, when the memory allocation routine malloc is called it returns a pointer to the type that was requested, which might be an array of characters or a structure. When the memory deallocation routine free is called the pointer may be of any type. A generic pointer is formed using the void type as the base type. For example:

```
extern void *malloc(unsigned );
extern int free(void *);
```

Prototype Induced Cast

The prototype function declaration provides more information for the compiler about a function and subsequent calls to it. If an argument in a function call is not the type declared, but may be cast to the correct type, then the compiler will perform the needed cast. The compiler will not flag the error if the cast can be performed. This example shows how the compiler will cast an integer to a long in an fseek call.

```
extern int fseek(FILE *, long, int);
int ret;
FILE *fd;
   ret = fseek(fd, 15, 0);          // What is written
   ret = fseek(fd, (long)15, 0); // What is understood
```

Prototype - char and float

Prototypes also allow the compiler to produce smaller code using the additional information about the function and arguments passed to it. If a function is declared to have a character or float argument type, then the compiler may choose not to extend the char to an int or the float to double. This saves instructions each time the function is called. The compiler must see the function prototype, both before the call to the function and before compiling the actual function, for char and float types to be handled correctly. Additional examples of prototyping are included in the library header files.

Autoprototyping

Autoprototyping is a process where if a prototype is not specified for a function, the compiler generates one according to how the function is used. Subsequent uses are checked against the generated prototype. This can be an invaluable aid to finding obscure bugs. Prototypes are generated using the default conversion rules. Thus, for

```
{  char c;
   unsigned char uc;
   signed char sc;
   int i;
   unsigned u;
   long l;
   unsigned long ul;
   float f;
   double d;
   char *p;

   func2(c,uc,sc,i,u,l,ul,f,d,p);
}
```

the prototype for func2 is equivalent to:

```
int func2( int,unsigned,int,int,unsigned,
           long,unsigned long, double,double,char *);
```

and is the same for:

```
func2(c,uc,sc,i,u,l,ul,f,d,p);
char c;
unsigned char uc;
signed char sc;
int i;
unsigned u;
long l;
unsigned long ul;
float f;
double d;
char *p;
{
   ...
}
```

which is **not** the same as:

```
int func2(char c, unsigned char uc, signed char sc,
      int i, unsigned u, long l, unsigned long ul,
      float f, double d, char *p);
```

for which the default conversions are not done. Functions that accept a varying number of parameters, such as printf() and scanf(), must be prototyped before they are used. Prototypes for all library functions are included in the appropriate .h files.

Compatibility With C

Every effort was made to maximize compatibility between Zortech C and Zortech C++ . Programs that previously compiled under Zortech C should compile and run without modification, with the following caveats:

- *In C++ , struct tag names, union tag names and enum tag names all are in the regular name space. In C, they are in a separate name space. This means that in C++ you cannot have tag names the same as variable names.*
- *C++ reserves several new keywords, so you cannot use those names as variable names.*
- *The C++ library should be linked before the C library. If you have add-on C libraries, they should work just fine, as long as they don't conflict with names in the C++ library, and are Microsoft compatible. (Earlier versions of Zortech C were Lattice Compatible).*

Please contact Zortech for details on how to upgrade your Zortech add-on libraries.

User-defined Type Conversions

You cannot define a user-defined type conversion to an unsigned type. For example:

now possible

```
struct
{  operator int();                          /* legal */
   operator unsigned int();                /* illegal */
}  stuff;
```

Compatibility with ATT C++

In Zortech C++ , shorts and ints are considered to be the same type. In ATT C++ , they are distinct. In ATT C++ , the constructors and destructors can be defined as returning int, even though they don't. The translator ignores the int. In Zortech C++ , there must be no return type specified for constructors and destructors.

In ATT C++ , a call to _entry() is inserted in the beginning of main() in order to call all the module constructors. In Zortech C ++ , this call is handled by the startup module, C.OBJ.

Zortech C++ regards declarations of global non-member functions with empty parentheses the same as in C (unknown number of arguments of unknown type). ATT regards these as functions with zero parameters.

C++ Programming Caveats, Bugs and Warnings

C++ is a young language, and there are many dark corners in it still for which there doesn't seem to be a good solution. Here are some of them:

- *Statics which are local to a function cannot have constructors or destructors.*

- *When declaring parameters to member functions, always use the fully prototyped method, as in:*

  int abc::def(int i) { ... }

  as opposed to:

  int abc::def(i) int i; { ... }

- *If you are assigning to 'this' within a constructor, make sure that you assign to 'this' along every path. This is of crucial importance if the class:*

  1. Has members with constructors
  2. Has virtual functions
  3. Is a derived from a class with any of the above

Code to watch out for is:

```
stack::stack()
{
   if (this == 0)
   {  this = my_allocator();
            // compiler inserts constructor calls here
   }
}
```

This should be written as:

```
stack::stack()
{
   if (this == 0)
   {  this = my_allocator();
            // compiler inserts constructor calls here
   }
   else
   {  this = this;                    // a dummy assignment
            // compiler inserts constructor calls here
   }
}
```

Note that this is a problem with C++ translators also, they just don't tell you about it!

There really isn't a 100% reliable way for the compiler to do this for you. The thing to remember is that the member constructor calls are always appended to any assignment to 'this'. If there aren't any assignments to 'this' in the constructor, the compiler generates one at the start, of the form:

```
if (this == 0)
this = _new((long) sizeof(*this));
            // compiler inserts constructor calls here
```

Mixing C and C++ Modules

Separately compiled C and C++ object files can be freely intermixed into one executable. It is not even necessary that the module containing main() be compiled with C++ . The caveats are:

- If you are not using the default libraries that are automatically inserted into the .OBJ file by the compiler, specify the C++ library to the linker before the C library.

- A C++ executable must be linked with a linker that will fix things up so that constructors and destructors are called for static variables. An alternative is to run the BUNCH program after the linker finishes. BUNCH is supplied with C++. It requires an up to date .MAP file in order to find the tables in the .EXE file. If the .MAP file is not current, BUNCH will probably trash your .EXE file. ZTC will generate a .MAP file and run BUNCH automatically if you give it the -B switch.

- Be careful when constructing the makefile so that the right compiler is run for each file.

- The default extension for C files is .c

- The default extension for C++ files is .cpp

- Be wary of how C++ generates global names for things like overloaded functions if you wish to call them from a C file. When in doubt, use OBJTOASM to look at the names in an object file, or examine the map file produced by the linker.

Miscellaneous

- There is no 'big' version of ZTCPP1. ZTCPP1 is already big!

- The module constructors are called by _entry(), which is called by the C.OBJ startup code. Thus, the commonly used kludge to put _entry() as the first statement in main() is not necessary. This call is always there, but the _entry() in the C library simply returns, while the _entry() in the C++ library calls the module constructors.

- The module destructors are called by exit(), so C++ and C have different implementations of exit(). exit() is called by the code in C.OBJ when main() returns.

- Pointers to the module constructors and destructors are stored in the global array _CTDT_PTR. The table is (unfortunately) a fixed size. If you wish to change the size, you can modify the file CTDTPTR.C (supplied), or you can simply include in one of your source files a definition for _CTDT_PTR[whatever]. The format of the table is explained in CTDTPTR.C.

- Remember that the _CTDT_PTR table must get filled in. The compiler can't do it because it doesn't know about other modules. There are two ways:

    1. Modify the linker to do it. Zortech LINK will do this automatically.

    2. Patch the .EXE file after it's linked. Zortech BUNCH will do this. If you use the Microsoft linker, BUNCH must be run on the .EXE file (the .MAP file is also required).

- To use CodeView, follow these steps, after making sure that Microsoft LINK appears on the PATH before Zortech LINK.:

  1. Compile source modules with -g flag (if you want line number information).

  2. Link with /co and /m flags. The Microsoft linker must be used. Zortech LINK silently ignores the /co flag.

  3. Run BUNCH on the output.

  4. Run CodeView on the result.

  5. If you use ZTC.COM, use:
     ZTC -g -co -B file...

- Zortech C++ makes use of advanced features of the .OBJ file format. In particular, it generates COMDEF records, and depends on the DOSSEG switch working correctly. Older Microsoft linkers will fail with these features. We recommend that you use the linker that comes with MASM 5.0 or later, if you wish to use Microsoft LINK. Zortech LINK works fine, but cannot handle CodeView, exepack options, overlays, or some of the more obscure options available with Microsoft LINK.

LINKING PROGRAMS

LINK.EXE

Linking programs for various memory models requires that you specify the appropriate library for that particular memory model. The library for each memory model can be seen in the following table:

Memory Model	C Library	FG-Graphics Library	C++ Library
S or T	ZLS.LIB	FGS.LIB	PLS.LIB
M	ZLM.LIB	FGM.LIB	PLM.LIB
C	ZLC.LIB	FGC.LIB	PLC.LIB
L	ZLL.LIB	FGL.LIB	PLL.LIB

NOTE: Care must be taken to ensure that object files compiled for different memory models are not linked together.

Neither ZTC nor ZED recognize the FGx.LIB, and these must be explicitly added to the link list when graphics functions are used.

LINK is a functional replacement for Microsoft's LINK. The reasons for this are:

- *There are many versions of MS-LINK, and many of the older versions of them will not correctly link Zortech C programs.*
- *Some older versions of MS-LINK will not work with the 16 byte pagesize library files.*
- *MS-LINK is no longer supplied with all copies of MS-DOS.*
- *LINK will do the .EXE file patching automatically that C++ needs. This eliminates the time-consuming extra pass needed (BUNCH) if MS-LINK is used.*

Using with ZTC

Ensure that LINK will be found in the path before MS-LINK is found. ZTC uses a common subset of the commands to both. ZTC automatically creates a linker response file if the command line to the linker exceeds the DOS line length limit.

Usage

```
LINK object files, exe file, map file,
                    library files [switches...] ;
```

object files object files separated with +'s

exe file name for the resulting .EXE file

map file name for the .MAP file

library files library files and directories separated with +'s

switches

/CO[DEVIEW] Ignored. for information on linking for codeview, see the section on Debugging with CodeView.

/CR[OSSREF] Write cross ref. file to "mapfile.CRF".

/CT[DTTABLESIZE]:nn
Set size of _CTDT_PTR table to nn (256 default).

/DO[SSEG] Special segment ordering used by Microsoft high-level languages.

/HE[LP] Causes LINK to display a list of current available options.

/I[NFORMATION]

Print info as linker executes including the names of the object files being linked, the order, and location.

/LI[NENUMBERS]

Include line numbers and associated addresses in .MAP file.

/M[AP] Write a full map file.

/MO[REINFORMATION]

Print more info as linker executes.

/NOC[TDTWARNING]

Don't warn when _CTDT_PTR table overflows.

/NOD[EFAULTLIBRARYSEARCH]

Ignore libraries specified in .OBJ files.

/NOI[GNORECASE]

Become case sensitive.

/ST[ACK]:nn Set stack size of exe file to nn.

Environment Variables

LINK uses the environment variable TMP for the path to where to write its temporary files. Object files are searched for in the current directory then in the LIB environment path. Library files are searched for in the current directory, then directories specified, followed by the LIB environment path.

The commands to LINK can be placed in a file (called a linker response file). The command to LINK is then:

```
LINK @file.lnk
```

where file.lnk contains something like:

```
FILE1+FILE2+
FILE3
EXEFILE
MAPFILE
LIBFILE/M
```

which is equivalent to:

```
LINK file1+file2+file3,exefile,mapfile,libfile/m
```

Differences between LINK and Microsoft LINK (MS-LINK)

- There is no 'interactive' mode for LINK.
- There can be no embedded spaces in LINK command lines.
- LINK does not recognize spaces separating object files to be linked. A plus sign (+) must separate object filename pairs.
- LINK does not require a ; after the command, but it ignores one if it is there.
- LINK does not do overlays.
- LINK does not have an option to pack .EXE files.
- LINK does not generate codeview compatible executables.
- LINK will recognize a C++ object module, and will patch the _CTDT_PTR[] table accordingly. MS-LINK won't do this.
- LINK generates slightly smaller .EXE files than MS-LINK does (it merely eliminates some useless padding in the header). The only effect this seems to have is that Microsoft's EXE2BIN program fails when trying to convert LINK output. The solution is to use Zortech's EXE2BIN, which will work on both the output of LINK and the output of MS-LINK.
- LINK doesn't support some of the more obscure MS-LINK options.
- LINK searches libraries for unresolved externals in each library in the order that the libraries are specified. LINK starts over at the beginning of the list of libraries for each remaining unresolved external. The first match it finds, it uses. The MS-LINK documentation states that MS-LINK does this also, but the behavior of MS-LINK suggests that it uses a 'best fit' approach.

For error messages see APPENDIX.

BUNCH.EXE

Synopsis

BUNCH reads a .EXE file and corresponding .MAP file. It patches the _CTDT_PTR table in the .EXE file to contain the function pointers for the static constructors and destructors. BUNCH is necessary if one or more modules in the .EXE file are C++ modules, and the .EXE file was linked with Microsoft LINK. If Zortech LINK was used, BUNCH is not necessary since the patching is done automatically by LINK.

Using with ZTC

The -B switch to ZTC will cause BUNCH to automatically be run after LINK is run.

If the -co switch is specified, and at least one of the source files to ZTC has a .CPP extension (indicating a C++ source file), then BUNCH is run because only MS-LINK can generate codeview compatible .EXE files.

Usage

```
BUNCH [-m[smcl]] [-tNNN] exefile
```

-m[smcl] memory model used (s is the default)

-tNNN set _CTDT_PTR table size (default = 256 function pointers)

exefile .EXE file (.MAP file is derived from this name)

.COM files

BUNCH must be run on the .EXE file **before** the .EXE file is converted to a .COM file with EXE2BIN. Use the -ms switch for this case if you must (it is the default).

Environment Variables

BUNCH uses the environment variable TMP to find the path to where its temporary files are to be written.

Example

To patch the program prog.exe, which uses the C memory model:

```
BUNCH -mc prog
```

Bugs

If BUNCH detects an error while patching the .EXE file it will delete the .EXE file since it has probably corrupted it. If the .MAP file corresponding to the .EXE file is not the one generated at the same time as the .EXE file, BUNCH will probably trash the .EXE file.

EXE2BIN.COM

EXE2BIN converts .EXE files to .COM files. The syntax is:

```
exe2bin file.exe file.com
```

Each filename must be fully spelled out.

Example:

To create a .COM program out of the object file hello.obj:

```
link ct+hello,hello;
exe2bin hello.exe hello.com
del hello.exe
```

Integer Only Programs

A significant saving in program size can be achieved if no floating point calculations are used in your program. When linking, link in the file INT.OBJ. This file prevents loading in the floating point routines from the library. If any attempt is made to perform a floating point operation the message

Floating point not loaded

is printed. Using the -mi switch will cause INT.OBJ to be linked in.

Examples

1.

```
LINK myprog+int;
```

Link myprog integer only.

2.

```
ZTC -msi myprog
```

Compile and link myprog using small model integer only.

Creating .COM Programs

A .COM file is a memory image of the program that is to be run. An .EXE is more complicated, containing relocation information that tells MSDOS how to load it. .COM files are smaller than .EXE files and load faster. To create a .COM file, use the CT.OBJ start-up object module when linking. The sum of your program's code, plus statically allocated data, must be less than 64k bytes. Run the resulting .EXE file from the linker through the utility EXE2BIN to create a .COM file. All this can be done with ZTC if the -mt switch is used.

NOTE: Ignore the warning about no stack segment from the linker. If you use Zortech LINK, you must also use Zortech EXE2BIN, not Microsoft's. Microsoft's EXE2BIN will create a corrupted program. Zortech EXE2BIN is compatible with the output of both Microsoft LINK and Zortech LINK.

PROGRAM EXECUTION

```
PROGNAME [=nnnn] {argument}
```

When the program is run, the following things happen:

1. The command line is scanned. If =nnnn is found, nnnn is taken to be the amount of space in bytes to reserve for the stack. The =nnnn is removed from the command line. The remainder of the command line is broken up into values for argc and argv[]. Spaces and tabs separate the arguments. Spaces and tabs can be embedded in arguments if the arguments are enclosed by " or '. The quotes are stripped off prior to inserting the argument in argv[]. The I/O redirection commands <, >, >> and | are handled by MS-DOS before the program ever sees them.

2. The stack is allocated, using nnnn if supplied else the value of the _stack global.

3. It is determined if an 8087 is present. or 287, 387

4. The function _entry() is called. In C++, _entry() calls the static constructors. In C, _entry() does nothing.

5. _main(argc,argv) is called. Normally, _main() just calls main(). If, however, _main.obj is linked in, _main() will perform wildcard expansion on argc and argv and pass the revised version to main(). _mainX()

6. When _main() returns, exit(0) is called. exit() handles calling the static destructors (for C++) and the flushing and closing of files.

7. When exit() is done, it calls _exit() which returns to MS-DOS. This is all explained in exhaustive detail in the file C.ASM on the library source disk, available separately.

Command Line Argument Expansion

Command line arguments, by default, are passed to the function main() exactly as defined on the command line. On a UNIX system the command interpreter expands file name arguments that contain wildcard characters ('*' or '?') to all file names that match the file specification. For example:

```
prog *.c
```

would cause a UNIX program to have an entry for each file with a .c extension in argv. The "*.c" would not be anywhere in argv.

ZORTECH C programs can have the file names expanded by using the _MAIN.OBJ routine to expand file name arguments before main is called. The object file _MAIN.OBJ must be included in the link step to perform the argument expansion.

The link command could be as follows.

```
LINK _MAIN+ PROG, PROG;
```

Using ZTC the command would be -

```
ZTC PROG.C  _MAIN.OBJ
```

When main(argc, argv) is called, argc will equal the total number of files matching the file specification with wild cards. argv will contain an entry for each file name, but there will not be a file specification containing wild cards. For example:

```
PROG ABC *.C "*.h" '*.obj'
```

is presented to main() as if it was:

```
PROG ABC a.c b.c "*.h" '*.obj'
```

Note that wildcard expansion was not done if the argument was protected by single or double quotes.

Controlling Stack Size

The stack size can be controlled in the following ways:

1. By using the '=stacksize' command line parameter when running the program.

2. By providing the declaration

```
int _stack = nnnn;
```

somewhere in the program source code. 'nnnn' is the number of bytes desired for the stack.

3. By modifying the C.ASM start up module.

NOTE: Do NOT use the /STACK parameter to the Linker to set the stack size.

The default stack size is set to 2048 bytes.

Use stack overflow checking if you are suspicious of stack overflow errors. Symptoms may be mysterious crashes. Compiling the program with the -s option causes stack overflow checking to be performed upon entry to each function.

GLOBAL OPTIMIZER

INTRODUCTION

What is a Global Optimizer?

A Global Optimizer (as opposed to a local optimizer) analyzes and optimizes the entire code for a function as a whole. Typical local optimizers only look at one statement at a time, or at short sequences of statements. A global optimizer uses a powerful technique called Data Flow Analysis (DFA) to gather information about the function. The more known about a function, the more optimizations are possible. This is why global optimizers improve code more than local ones can.

Why is it needed?

It is true that most C++ programs can be tuned to a particular machine and that the global optimizer does few transformations that cannot be duplicated at the source level. However, this usually does not happen because:

- The goals of ease of maintenance and legibility can be more important than cryptic tuned code.
- Code that is tuned for one machine may turn out to be badly tuned for another (the most obvious example of this is varying support for register variables in different compilers).
- Tuning code requires a lot of the programmer's time. Software is meant to allow programmers to be more productive, and a global optimizer largely eliminates the need to tune programs.
- The C++ parser uses simple methods to expand inline functions, relying on the global optimizer to 'fix' them into more acceptable code.

When do I use it?

The Global Optimizer provided here executes very slowly relative to the speed of the other two passes. Therefore, it should only be used on routines that are already debugged. It is also not useful on throw-away programs. It is most useful on frequently executed programs. The rule of thumb is: if the number of executions of the program multiplied by the speedup is greater than the time spent optimizing, it is worth it.

How much performance speedup can I expect?

Typical compute bound programs will speed up between 0 and 30%. I/O bound programs may not speed up at all. The most dramatic speed improvements will be in small, but frequently executed, loops. The least speed improvements will be in code that consists mostly of function calls.

Does the Global Optimizer make assembly code unnecessary?

No, optimized code still does not do nearly as well as carefully crafted assembly code. One reason for this is that the assembly language programer knows more about the algorithm than the compiler. The compiler will always take the conservative approach to optimization to insure that only safe optimizations are done. But many projects cannot afford the man-hours to carefully craft an entire project in assembly code, and an optimizing compiler then offers the most cost-effective solution.

What a Global Optimizer will not do

A Global Optimizer will not replace inefficient algorithms with efficient ones. For example, it will not recognize a bubble sort and replace it with a quicksort. Regardless of how good a compiler might be, usually the most dramatic increases in execution speed are achieved by improving the algorithm used.

Operation

Running the Optimizer with ZTC

To run using ZTC, use the -o switch. Note that the -o switch with a filename after it specifies the output file to ZTC! For example:

ZTC demo -o

Means compile, optimize, and link demo

ZTC demo -otest.exe

Means compile and link demo, then name the resulting file test.exe. The optimizer is not run.

ZTC demo -o -otest.exe

Means compile, optimize, then link demo, and name the resulting file test.exe.

Running Stand-Alone

ZTG is run on the output of ZTCPP1, ZTC1, or ZTC1B and the output from ZTG forms the input to ZTC2 or ZTC2B.

ZTG input [-ooutput] {(+|-)flags} [-v]

input	File that is the output from ZTCPP1, ZTC1 or ZTC1B. An extension of .TMP is assumed if one is not given.
output	Output file name for ZTG. If not specified, the output file name is formed by taking the input file name and appending the extension .TMO to it.
-v	Verbose compile (list function names as they are processed).

Other Flags

For the rest of the flags, + means turn on optimization, - means turn it off.

all	Perform all optimizations (default).
cnp	Constant propagation.
cp	Copy propagation.
da	Dead assignment elimination.
dc	Dead code elimination.
dv	Dead variable elimination. Compute live ranges.
gcse	Global common subexpressions.
li	Loop invariant removal.
liv	Loop induction variables.
loop	Iterate until no more optimizations.
none	Do no optimizations.
reg	Do our own register assignments.
space	Favor space optimizations over execution time optimizations.
time	Favor execution time optimizations over space optimizations. (+**time** is the default)
vbe	Very busy expressions.
w	Turn on warnings (like variable used before it's defined).

Examples:

```
ZTCPP1 demo
ZTG demo
ZTC2 demo
LINK demo;
```

means compile, optimize, and link the program demo.cpp.

```
ZTG demo +space -w
```

means favor code space optimizations, turn off warnings.

Advanced Information

Constant propagation (+cnp)

Consider the code:

```
A=5;
for(i=0; i<A; i++)
      abc[i]=A;
```

It is clear that A will always have the value 5 within the loop body. Therefore, it can be replaced with a 5:

```
A=5;
for(i=0; i<5; i++)
      abc[i]=5;
```

Constant propagation opportunities frequently occur when loop rotation is done.

Copy propagation (+cp)

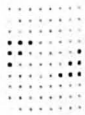
Copy propagation is similar to constant propagation, except that variables are copied rather than constants.

```
A=b;
for(i=0; i<A; i++)
        abc[i]=A;
```

is replaced with:

```
A=b;
for(i=0; i<b; i++)
        abc[i]=b;
```

Copy propagation will frequently uncover dead assignments that can be removed (such as the assignment to A).



Dead assignment elimination (+da)

Assignments to variables that have no further use are eliminated.

```
int abc()
{
   int f;
   static int g = 0;
   if (g!=3)
   {
      f=3;          /* this is a dead assignment to f */
      g=3;            /* this is not a dead assignment*/
   }
}
```

Dead code elimination (+dc)

Dead code is code that can never be reached, and so it is deleted. Sometimes dead code can be rather subtle:

```
#define VALUE 0
if (VALUE>10)
      a=dead+code;          /* this statement is dead */
```

Dead variable elimination, compute live ranges, do register allocation by coloring (+dv)

Dead variables are automatic variables declared but never used. They often result from other optimizations. A variable's live range is that portion of a function which its value must be preserved. If the live range is 0, the variable is dead! If two or more variables have non-intersecting live ranges, they can occupy the same memory location. This becomes interesting when allocating a large number of variables into a small number of registers. The 8088 has a small number of registers, each with special capabilities, so it is difficult to 'reserve' registers for register variables. So the live range of each register is computed instead, and compared with the live variable range. When they do not intersect the variable is assigned to that register. This is called 'coloring' and is analogous to coloring a graph of the variables, with the number of different colors being the number of registers, and trying to color it such that at any point no two of the same colors exist.

Global common subexpressions (+gcse)

Common subexpression elimination is the elimination of redundant computations.

Loop invariant removal (+li)

If an expression appears in a loop body, but its result never changes, then it is a loop invariant, and its computation can be removed from the loop.

```
while(f())                    /* b*c is loop invariant */
        g(b*c);
```

becomes:

```
T = b*c;
while(f())
        g(T);
```

Loop induction variables (+liv)

Loop induction variable replacement involves:

```
for (i=0; i<10;i++ )
   a[i*2+3]=5;
```

becomes:

```
for (T=&a[3]; T<&a[23]; T+=2)
   *T=5;
```

This is usually one of the most fertile areas for program speedups.

Iterate until no more optimizations (+loop)

Performing many optimizations frequently makes further optimizations possible. This flag tells the optimizer to continue until it can discover no further optimizations.

Do our own register assignments (+reg)

Use of the 'register' storage class is unnecessary, because the optimizer puts as many variables in registers as possible. Which variables have priority is determined by how often they are used and how deeply nested in loops they are.

Since the optimizer frequently generates new variables and eliminates the originals, an automatic register allocator is essential.

If this flag is used, however the optimizer will honor the register declarations in the source code.

Space/time tradeoffs (+space and +time)

Sometimes a choice can be made as to whether space or time optimizations should be done. One specific example is in the case of:

```
while (f())
      g();
```

Space optimization would give us:

```
goto L1;
do
{
   g();
LI:   ;
} while (f());
```

Time optimization would give us:

```
if (f())
{
   do
      g();
   while(f());
}
```

Very busy expressions (+vbe)

Very busy expressions are those that occur along more than one path in the flow graph:

```
if (condition)
      a=b*c;
   else
      d=b*c;
```

Here b*c is a very busy expression. The optimization is:

```
T=b*c;
if (condition)
      a=T;
   else
      d=T;
```

Note that this saves space only; not time.

Hints and Tips

Use of const and volatile keywords:

The *const* keyword states that the variable is never modified, and so the optimizer does not have to make worst case assumptions about it changing. *volatile* tells the optimizer that the variable can change asynchronously, and so loads and stores to it should not be 'optimized out'. *const* is useful for static tables. *volatile* is used for hardware locations (like video screen memory) and for data that could change when an interrupt occurs (such as DOS or BIOS variables, or flags set by an interrupt driver).

Examples:

```
const static int a=5, b[3]={1,2,7};
extern volatile bool flag;
```

What to do if your program works when compiled without the optimizer, but fails when compiled with it

The optimizer uses a rather strict interpretation of the definition of the C++ language, and strives to conform to it. There are several aspects of it that are undefined, and this undefined behavior may be different depending on whether or not the optimizer was run. Some reasons why code may behave differently when optimized:

1. The code is dependent on the order of evaluation of side effects.

Such as:

```
a=f1()+f2();
```

Whether f1() or f2() is executed first is undefined. The order of evaluation can be controlled by using explicit temporaries, such as:

```
tmp=f1();
a=tmp+f2();
```

Now the evaluation order is guaranteed.

2. The code is de-referencing uninitialized pointers. Since the optimizer rearranges the physical allocation of variables, unitialized variables may contain different values.

3. The code is storing data past the end or before the beginning of arrays or data allocated with malloc().

4. The code refers to data that was freed with realloc() or free().

5. A variable that could be modified at interrupt level is not declared with a 'volatile' storage class modifier.

6. There is a bug in the optimizer.

Unless the last case is true, recompiling code with the optimizer can uncover similar problems to that encountered when porting to another compiler. Adherence to standard 'portable' coding practices will minimize any problems encountered using the optimizer.

ADVANCED FEATURES

Memory Models

Zortech's default memory model is Small. If very large programs are to be written it is necessary to use a larger model. The segmented architecture of the 8086 processor family complicates addressing in large programs. Programmers of larger programs face trade-offs between efficiency and full addressability. These combinations of addressability and efficiency are reflected in the five memory models used by Zortech C++.

8088/86 Segmented Architecture

This processor has 16 bit registers, but 20 address lines which means it can address up to one megabyte of memory. It can be seen that an absolute address can't be held in one 16 bit register. To overcome this, segmentation is used. Memory is divided into segments of up to 64K (the maximum number that can be held in a 16 bit register). An absolute address can then be calculated by adding the OFFSET from the beginning of a segment to the address at the start of the SEGMENT.

However, the segment's start address must also be held in a 16 bit register. This is achieved by putting the highest 16 bits in the register and assuming that the last four are all zeroes. Thus memory addressing in the 8086 is performed by generating a 20-bit physical address from two 16-bit numbers called the Segment and Offset. The physical address is calculated by shifting the 16-bit segment left four bits and adding the offset to it. There are four special purpose segment registers: CS, DS, SS and ES.

CS is the code segment register, it always contains the segment of the currently executing code. DS is the data segment register, it usually contains the segment of all global and statically allocated data. SS is the stack segment register, it contains the segment of the hardware stack. ES is the extra segment register, it is used to address any segment. ES is typically used when dereferencing far pointers.

Five Different Models

ZORTECH C++ offers five possible memory models called the T model, S model, M model, C model, and the L model. These memory models are defined by their program code and pro-gram data space limitations. The M and L models use FAR calls (a call instruction that uses both a segment and an offset) for function calls.

These FAR calls can span up to one million bytes in memory, but are less efficient than a near call instruction. The C and L models use far pointers to reference data. These far pointers again can span one million bytes of memory, but they are less efficient than their near pointer counterparts. The following table shows addressing limitations of each model.

Memory Model	Code Space	Data Space
T	64K	64K *
S	64K	64K
M	1MB	64K
C	64K	1MB
L	1MB	1MB

* The total of code plus static plus global data must be less than 64K

Choosing a Memory Model

Selecting a memory model is a trade off between addressability and efficiency. Using the S (small model) which is the default, is the most efficient and is the right choice for most programs. Obviously, programs which take up more than 64 kilobytes of code or data require another memory model. Here is how to choose from among them:

1. If the compiled total of code for the program and its library functions will be greater than 64 kilobytes--then either the M model or the L model will be required.

2. If the total compiled program code does not go over the 64 kilobyte limit, then the S model or C model will be more efficient.

Pointers

A pointer is a variable that contains a physical address. There are two types of pointers, called a far pointer and a near pointer.

Near Pointers

Near pointers can be used to access any location within a 64 kilobyte segment. It requires only two bytes of storage to hold the offset and is thus more efficient in the amount of RAM needed to store it, and the time it takes to use it. It is assumed that segment register DS contains the correct segment value. Near pointers are the default in the T, S, and M memory models for data pointers. Near pointers are used for function pointers in the T, S, and C models.

Far Pointers

Far pointers require four bytes of storage because they are made up of a two-byte segment and a two-byte offset. The segment is a base address that will be put into the correct segment register and the offset provides an index off that base address. Far pointers are the default for data pointers in C and L models and function pointers in M and L models.

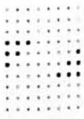
Macros for Use with far Pointers

The following macros allow you to determine whether the pointers are to be near or far.

```
#define SPTR 1        /* 1 if data pointers are 2 bytes
                                                else 0 */
#define LPTR 1        /* 1 if data pointers are 4 bytes
                                                else 0 */
#define LCODE 1         /* 1 if M or L Models (function
                        pointers are 4 bytes) else 0 */
```

See also the section on *Predefined Macros*.

The macro MK_FP (seg,off) creates a far pointer from a segment and an offset. MK_FP, FP_SEG, and FP_OFF are described in *Part Three - The Function Library*.



Normalizing Far Pointers

A far pointer is 'normalized' if its associated offset is less than 16. This means that an absolute address could be calculated by taking the segment part of the pointer and appending the 4 least significant bits of the offset. If far pointers are normalized they can be compared, whereas if they are not normalized then two far pointers could point to the same absolute address but have different values in their registers. For example:

SEGMENT	OFFSET	ADDRESS
0x1234	0x0098	0x123D8
0x123D	0x0008	0x123D8

The second example above is already normalized. The offset contains a 1 digit hex number. i.e. less than 16 decimal. Far pointers can be normalized by calling the library function _farptr_norm(). For example:

```
#include <dos.h>
char far*p;                    /* p is non-normalized */
p = _farptr_norm(p);           /* p is now normalized */
```

Far Pointer Arithmetic

Calculations involving far pointers are very different from calculations using near pointers. The difference is due to the two components that make up a far pointer. They must be understood to be effectively used. Some operators generated by the compiler employ both the segment and offset during their operation. On the other hand, some use only the offset.

For operators which use only the offset portion of the long pointer, a segment wrap can occur. This occurs when the offset reaches its maximum value and then wraps around to zero. Operators that affect only the offset portion are:

```
+-    +=    -=    ++    >
<     >=    <=    --
```

Operators which work on the full 32-bit value are:

```
==    !=    !
```

These operators compare both components of the far pointer, but they do not perform normalization before the operations are performed. As a result, the == will return a zero for two pointers which point to the same location, but have different segment values. You can avoid this problem by normalizing the pointers before performing the == comparison. For example:

```
#include <dos.h>
char *p1, *p2;
   if (_farptr_norm (p1) == _farptr_norm (p2))
```

For most programs the above operations will not cause any problems. Full 32-bit arithmetic can be performed on long pointers by:

1. Converting the far pointer to a long int via _farptr_tolong()
2. Performing the operation(s)
3. Converting it back to a far pointer via _farptr_fromlong()

For example, to add 100,000 to a pointer you could do the following:

```
#include <dos.h>
char *p;
p = _farptr_fromlong(_farptr_tolong(p)+100000);
```

A far pointer may be initialized to a specific hardware address by using MK_FP(). For example, to create a pointer to an IBM-type computer's monochrome display memory:

```
#include <dos.h>
main()
{
   unsigned *p;
   int i;
      p = MK_FP(0xB000,0);          /* display memory */
      for(i=0; i  <  80*25; i++ )
            p[i] = 0x0700+' ';
}
```

Using near and far Keywords

The default pointer type for data pointers can be overridden by using the near or far keywords. Declarations look like:

```
int far *p;          /* p is a <far pointer to><int> */
int * far *p;                /* p is a <far pointer to>
                                  <pointer to><int> */
```

The near and far keywords are right-associative, that is, they are attached to the * on their right. This is in contrast to the const and volatile keywords, which are left-associative. Yes, this is confusing, more examples are in order:

```
                          /* declare p as <pointer to>
              <volatile near pointer to> <const int> */
int const near * volatile *p;
```

Remember that <pointer to> becomes <near pointer to> for the T, S and M models, and <far pointer to> for the C and L models. The near and far keywords are not ANSI sanctioned, they are extensions to support the segmented architecture of the Intel CPUs. If the -A (ANSI compile) compiler switch is used, these keywords will not be recognized. In any case, we recommend that you avoid using the identifiers near and far for variable names.

Using Near Pointers

Near pointers can be used in the C and L models only if the data pointed to is addressible by the DS register, since the code generator assumes that near pointers point into the data segment. This means that near pointers can only be used to point to static or global data, not automatic (stack) data or function parameters. The primary usefulness of near pointers is to speed things up in the C and L models.

Using Far Pointers

Far pointers have many more uses. Typical uses would be to access video memory or BIOS data areas directly. For example, code to clear the monochrome text screen might be:

```
#include <dos.h>
unsigned far *vp,u;                    /* Build pointer to
                                       start of display */
vp = MK_FP(0xB000,0);                  /* Clear display */
for (u = 0; u < 80*25; u++)
   vp[u] = 0x0700 + ' ';
```

Converting Between Near and Far Pointers

A far pointer can be converted to a near by:

```
#include <dos.h>
char near *np, far *fp;
np = (char near *) FP_OFF(fp);
```

Note that FP_SEG(fp) must equal DS!

Near pointers can be converted to far by:

```
#include <dos.h>
char near *np, far *fp;
fp = MK_FP(getDS(),(unsigned) np);
```

Note how the value of DS is obtained for building into the far pointer.

Alignment Suppression within Structures

The default alignment of structure members is on word boundaries except where two or more single character variables follow each other, when the first character will be aligned on a word boundary, whereas subsequent ones will be byte aligned. Alignment within structures can be suppressed so structure members are aligned on byte boundaries. This suppression is useful for defining a struct that will map onto a hardware device or a predefined data element like an MSDOS directory entry.

The alignment suppression is only valid within structures; everything else will still be aligned on word boundaries. Each structure in an array of structures compiled with the -a option will start on a word boundary.

Example

Offsets in Bytes		
	Normal Alignment	Suppressed Alignment

struct dat
{
char name[5];
int age;
char sex;
char street[7];
long height;
};
	
	0
	6
	8
	10
	18
		
		0
		5
		7
		8
		15

sizeof(struct dat)	22	19

NOTE: Each file referencing a structure must be compiled with the same type of alignment. If two files reference the same structure, one compiled with the -a option and one not, any erroneous results could be very hard to debug.

Debugging C++ Programs

Using Debuggers

Programs compiled with Zortech C++ can be debugged using third-party debuggers that require the source code line numbers. The C++ compiler provides line numbers in the object file when the program is compiled with the "-g" option. When this option is used the size of the object file increases significantly. Refer to your debugger manual for linking and debugging the resulting program.

Other Aids

There are some features added to the run-time library to aid in detecting common run-time errors. These are:

If a NULL pointer is passed to printf where a pointer to a string is expected, printf uses the string "(null)" instead. For example:

```
printf("String '%s'\n",NULL);
```

will print:

```
String '(null)'
```

If a NULL near pointer is dereferenced, the string that is at location 0 in the data segment is "ERROR: NULL pointer".E.g:

```
char buf[100];
strcpy(buf,NULL);
printf("String '%s'\n",buf);
```

will print (for T, S and M models):

```
String 'ERROR: NULL pointer'
```

Debugging with CodeView

The Microsoft CodeView debugger is a line oriented source debugger. To obtain best results using this debugger, a few rules outlined below should be followed.

- #include files should be limited to #define macros and type definitions as CodeView is not able to trace through these files.
- Because of the line oriented nature of CodeView, each statement in your source code should occupy a separate line.
- CodeView does not break down complex macros so if a macro is suspected, write it as a regular source statement.
- Compile source modules with the -g flag.
- Link with /co (CodeView) and /m (map) flags. The Microsoft linker must be used.
- Run BUNCH on the output, if there are any C++ modules in the executable.
- Run CodeView on the result.

If you use ZTC.COM, use:

ZTC -g -co -B file...

Be sure that Microsoft LINK appears on the path before Zortech LINK.

Note that Codeview compatability is limited to global variables and line numbering only. Local (automatic) variables cannot be observed.

Refer to your Microsoft CodeView manual for further information on debugging.

TOOLS

MAKE

The ZORTECH MAKE utility is inspired by the UNIX utility of the same name. it is a full-featured MAKE utility for MSDOS. Features include macros, full dependency checking, MSDOS internal commands, and descriptive error messages.

What is MAKE?

MAKE is a utility that automates program maintenance. Instead of explicitly typing in commands to compile and link your program(s), it is possible to write all of the steps necessary in a text file called MAKEFILE then just give the command MAKE when you wish to compile and link.

This is particularly useful for programs built up from many source files, as it will only recompile the source files which have been updated. It does this by looking at the date and time stamp of each relevant file before carrying out any tasks. This implies that if you are using MAKE, it is necessary to ensure that your machine always has the correct date and time set. MAKE carries out the tasks of building a new program version as defined in a description file. For example, the steps to build a program from a single C++ source file would be:

1. **Compile the source program into an object program.**

2. **Link the object program with the C++ library.**

MAKE differs from batch processing by performing only the required actions, while a batch file will execute all tasks, whether they are needed or not.

Getting Started With MAKE

This section will guide you through the initial set-up of MAKE and show an example session.

The MAKEFILE

The description file that describes the actions needed to build your program is called the MAKEFILE. This the default name for the description file and also what the description file will be called in this manual. The MAKEFILE is a text file created with a text editor (e.g. ZED). It contains one or more *dependence lines,* each followed by one or more command lines. The dependence line has a target followed by a colon and a list (possibly null) of dependent files, i.e. files needed to create the target. Command lines follow a dependency line and must begin with a tab or space character. These lines are executable command lines just like those you would type at the MSDOS prompt.

```
TARGET-FILE : DEPENDENT-FILE DEPENDENT-FILE2
              COMMAND 1
              COMMAND 2
```

An Example MAKEFILE

```
HELLO.EXE:  HELLO.OBJ          #  Target / Dependence line
            ZTC HELLO.OBJ      #  Command / Rule line

HELLO.OBJ:  HELLO.CPP          #  Target line
            ZTC -c HELLO.CPP   #  Command line
```

This shows how to create a MAKEFILE for a program written in one source file only. In the above example the target is HELLO.EXE which depends on HELLO.OBJ which in turn depends on HELLO.CPP. MAKE first checks if the date and time of HELLO.CPP is more recent than HELLO.OBJ. If it is more recent, or if HELLO.OBJ does not exist, then the command line ZTC -c HELLO.CPP is carried out. This will create a new HELLO.OBJ which means that this will now be newer than any existing HELLO.EXE so the command ZTC HELLO.OBJ will also be carried out. To use the above MAKEFILE you would just type MAKE.

Dependency files

Dependency files are all the source files that, when changed, cause a change in their resulting target files. For example, modifying a C++ source file will cause the object file resulting from the compilation to change. But the source file may not be the only dependency. When the source file includes other files, then these included files are also dependency files. An example would be the line

```
#include "DEFS.HPP"
```

in a source file prog.cpp. This would MAKE the file defs.hpp a dependency. The MAKEFILE line would be:

```
PROG.OBJ: PROG.CPP DEFS.HPP
```

which means that PROG.OBJ depends on PROG.CPP and DEFS.HPP.

Second Example

A MAKEFILE may contain comments to describe what MAKE is to do. Comments are started with a '#' character and continue until the end of line.

```
# MAKE file for building the dump utility program

dump.exe :  dump.obj            # first target line
            ZTC dump.obj        # first command line

dump.obj :  dump.cpp stdio.h    # target is dump.obj
            ZTCPP1 dump         # compile command
            ZTC2 dump           # Pass 2 of compiler
```

This MAKEFILE will first cause dump.cpp to be compiled if it is newer than dump.obj or if dump.obj can't be found. Next, dump.obj will be linked if dump.exe does not exist or if dump.obj is newer than dump.exe. If dump.cpp was recompiled in the first step of MAKE, then dump.exe will have to be linked because dump.obj is newer. MAKE will then end and return to MSDOS.

A MAKEFILE with More than One Source File

```
EXAM9A.EXE :   EXAM9A.OBJ EXAM9B.OBJ       #Target line
               ZTC EXAM9A.OBJ EXAM9B.OBJ   #Link
               EXAM9A                      #Run program

EXAM9A.OBJ :   EXAM9A.CPP                  #Target line
               ZTC -c EXAM9A.CPP           #Compile C++
                                            module

EXAM9B.OBJ :   EXAM9B.ASM                  #Target line
               MASM EXAM9B;                #Assemble
                                          assembler module
```

The above example will first of all cause MAKE to compare the dates/times of exam9a.obj and exam9a.cpp and compile if necessary. The second step will be to compare the dates/times of exam9b.obj and exam9b.asm and assemble if necessary. If either of these steps were performed or if either object file is newer than any existing exam9a.exe then linking will take place.

If exam9b had been written in C++ and not assembler then the last two lines of the MAKEfile would have been -

```
EXAM9B.OBJ: EXAM9B.CPP              # Target line
            ZTC -c EXAM9B           # Compile
```

Running MAKE

MAKE is run by typing "MAKE" in response to the MSDOS command prompt. MAKE will then read the default MAKEFILE for the dependency and commands it is to perform. Using the previous example, the following program output shows what MAKE does in order to build the file dump utility.

```
B > MAKE
ZTCPP1 dump
ZTC2 dump
ZTC dump.obj
BLINK 2.12 Copyright (c) 1986-88 by Zortech, written by Bjorn N. Freeman
BLINK complete. Time: 5.45 seconds

B >
```

If you then try to re-MAKE the program -

```
B > MAKE
Target file "dump.exe" is up to date

B >
```

MAKE Command Line Usage:

```
MAKE {target} {macro=text} {-dilnqst} [-fMAKEfile]
```

Switches:

target	What targets to MAKE
macro=text	Define macro to be text. This overrides any definition in the makefile.
-d	Output debugging info
-ffile	Use file instead of MAKEFILE
-f-	Read makefile from stdin
-i	Ignore errors from executing MAKE rules
-l	Show macro expansions in makefile
-n	MAKE rules are not executed, just written to stdout
-q	Return with errorlevel 1 if rules would be executed
-s	Do not echo MAKE rules
-t	Just touch files that are out of date.

Uses of the MAKE Arguments

The MAKE target is the first target file defined in the MAKEFILE unless it is defined on the command line. This can be used with a MAKEFILE that builds several utility programs when only one is wanted. Some MAKEFILEs not only build programs, but test and install them too. Specifying the target in this case would allow a user to just build, just test, or just install the new program. Defining macros on the command line provides ways to modify the execution or meaning of what MAKE does.

User Defined Macros

MAKE supports user defined macros. A macro is a string of one or more characters that is defined once in a MAKEFILE. When the macro is encountered in the remainder of the dependence lines and command lines, it will be replaced by its definition. Macros are defined with lines of the form:

MACRO = macro_definition

The macro definition includes all text between the = character and the newline. A macro definition may be continued on another line by ending the first line with a \ . Macros are invoked by a dollar sign and the macro name enclosed in parentheses. Single character macro names do not require parentheses. For example:

```
$(MACRO)
$A
$(ONE_TWO)
```

Single character macro names do not need () when expanded.

Definitions for macros are searched for in the following order:

1. Definitions from the command line.
2. Definitions in the MAKEFILE.
3. Definitions from the environment.
4. The macro is expanded to nothing.

Pre-Defined Macros

MAKE has macros that are not defined in the MAKEFILE, but are predefined by the MAKE program. These macros are:

$? List of dependencies that are newer than target

$** Full list of dependencies

$* Name of current target without extension

$< From name of current target, if made using an implicit rule

$@ Full target name

$$ Expand to $

Rule Flags

In order for MAKE to have full access to all the MSDOS features, three rule flags to tell MAKE how to execute the commands provided. The rule flag appears just before the command.

+ Force use of COMMAND.COM to execute the command. This is useful for executing batch files, and to execute rules that include pipes or I/O redirection.

- Ignore exit status from this rule. This causes MAKE to continue executing even if an error occurred during this command.

@ Do not echo this rule when it is executed.

Example

```
install:
    -rm \bin \prog
    +prog >file
```

MS-DOS internal commands

The following commands are internal to MS-DOS, that is, when MAKE executes one it does so by executing COMMAND.COM and passing the line to be executed to it.

break	echo	rmdir
cd	erase	ren
chdir	exit	rename
cls	if	time
copy	md	type
ctty	mkdir	ver
date	pause	verify
del	rd	vol
dir	rem	

Special targets

.SILENT Same as if -s was specified on command line. Do not put a : after this.

.IGNORE Same as if -i was specified on command line. Do not put a : after this.

.DEFAULT Rules following this target are executed for targets that have no rules specified for them.

Logging

For long and verbose MAKEs, the output of MAKE along with all messages can be redirected to a log file for later perusal.

```
MAKE >log
```

Note that only stdout output of programs run by MAKE are redirected. ZTC.COM, ZTCPP1.EXE, etc. all redirect fine. MASM 4.0 and earlier versions won't.

Implicit Rules

Rules can be defined to inform MAKE how to create new files implicitly. A commonly used rule tells MAKE how to create an object file (.OBJ) from a C ++ source file (.CPP). Once the rule is defined only dependencies need to be defined and MAKE will know how to build the target.

Implicit rules are defined by using the source file name extension immediately followed by the destination file name extension. Predefined macros are used in the commands to describe the file names. For example, the rule to build an object file from a C++ file is:

```
.cpp.obj :
        ZTC -c $*
```

A simple MAKEFILE would then be -

```
.cpp.obj :                          # no dependencies
            ZTC -c $*               # general rule

hello.exe:  hello.obj               # target line
            ZTC $*.obj              # link hello.obj

# target line for hello.obj is unnecessary
# as MAKE now knows how to create an .obj file
```

Another common rule is used to create an object file from an assembly source file. This is done with the following rule:

```
.asm.obj :
        MASM $*;
```

Creating Linker Response Files

MAKE can create linker response files automatically. A linker response file is used by the linker for determining the files to link (see your linker manual). The advantage of having MAKE create the linker response file is only having to edit the MAKEFILE when you add or delete files from your program. Otherwise, both the MAKEFILE and the linker response file must be edited when you change how your program is built.

The linker response file is made using the MSDOS ECHO command by ECHOing each line into the linker response file. The output of the ECHO command must be redirected to the response file using the " > " and " >>" redirection specifiers. The first line must be made with a " > " so that the file is created (or truncated to zero length). All additional lines are ECHOed using the " >>" redirection specifier. For example:

```
#
# MAKE file for linking only!
#

myprog.exe: file1.obj file2.obj
            link @linker.rsp

# Create a new linker response file
# whenever MAKEFILE is changed

linker.rsp :   makefile
               echo file1.obj+file2.obj > linker.rsp
               echo myprog.exe  >>linker.rsp
               echo myprog.map  >>linker.rsp
               echo mylib.lib  >>linker.rsp
```

Summary of the differences from the UNIX MAKE

The ZORTECH MAKE program was designed to be as compatible with the UNIX MAKE as is practical in the MSDOS environment. Fundamental differences between MSDOS and UNIX have caused differences in ZORTECH MAKE, and assumptions that can be made in the UNIX environment cannot be made for MSDOS.

- No default rules are defined for ZORTECH MAKE.
- Many UNIX shell commands may differ or have no analog under MSDOS.
- The maximum command length is 127 characters after macro expansion.
- Since '\' is used as a path separator in MS-DOS be wary of things like:

```
LIB=\zortech\
```

because the last \ will cause a line continuation.

For error messages see APPENDIX.

TOUCH

TOUCH causes the last modified date and time of the files on the command line to be changed to the current date and time. The format of the command is :

```
TOUCH file1 file2 ...
```

TOUCH is used in conjunction with MAKE for maintaining the dates of files and modifying the operation of MAKE. For example, if you want all the C++ source files of a program to be recompiled instead of the ones last modified, then the following commands would do it:

```
TOUCH *.CPP
MAKE
```

Also, if one or more files were edited, but the only changes made to the files were comments or other things that would not affect the resultant object file, then the object files could be "touched" so MAKE would not re-MAKE them needlessly.

```
TOUCH *.obj
MAKE
```

OBJTOASM

OBJTOASM disassembles .OBJ files. If there are line number records in the .OBJ file, OBJTOASM will show mixed source/assembly in its output. The output may be redirected to another file.

Usage

```
OBJTOASM file [sourcefile]
```

The .OBJ extension is assumed. If a sourcefile is specified, OBJTOASM assumes that that is the source file that line number records in the .OBJ file refer to. Otherwise, OBJTOASM uses the source file name that is specified in the .OBJ file.

Example

To compile file.cpp and place the resulting assembly code in file.asm:

```
ZTC -c -g file
OBJTOASM file  > file.asm
```

Bugs

Some instructions are not decoded properly (like 8087 instructions). The output cannot be assembled without some hand editing. Data segments are ignored.

INTERFACING WITH ASSEMBLER

This section of the manual assumes that the reader is familiar with the 8086 processor, 8086 assembler and C++ . Subroutines can be written in assembly language and linked in with a C++ program. Assembly functions are normally written because:

- ***A subroutine must be as fast as possible.***
- ***Not all the capabilities of the computer or the instruction set are available from the C++ language or its library functions.***
- ***To provide an interface to functions compiled with another compiler.***

Program Sections

The code and data in the object file produced by the compiler are grouped into sections. The horizontal bars delineate areas of addressability by segment registers. These sections have names and properties as follows:

T Memory Model (.COM Programs)

Note that Zortech .COM files are not the same as in other compilers. In other compilers, CS==SS==DS for .COM files, and the entire size of the program plus stack and heap must be less than 64k. In Zortech .COM files, only the code size of the code plus DGROUP areas must be less than 64k. Considerably larger .COM programs can thus be created. Also, the only difference between a T Model Program and an S Model program is how it is linked.

T Memory Model (.COM Programs)

Note	Offset	Contents	Segment Name	Class	Group	Segment Register
1	0x000	PSP				CS
2	0x100		_TEXT	CODE	CGROUP	
3			C_ETEXT	ENDCODE	CGROUP	
5			NULL	BEGDATA	DGROUP	DS,SS
6		inited data	_DATA	DATA	DGROUP	
7			CONST	CONST	DGROUP	
8	_edata	uninited data	_BSS	BSS	DGROUP	
15		common data	c_common	BSS	DGROUP	
16	_end	not used	STACK	STACK	DGROUP	
11		stack				
17		heap				
13		far heap				

S Memory Model

Note	Offset	Contents	Segment Name	Class	Group	Segment Register
1	0x000	PSP				
2			_TEXT	CODE		CS
3			C_ETEXT	ENDCODE		
4		Other segments				
5			NULL	BEGDATA	DGROUP	DS,SS
6		inited data	_DATA	DATA	DGROUP	
7			CONST	CONST	DGROUP	
8	_edata	uninited data	_BSS	BSS	DGROUP	
9		common data	c_common	BSS	DGROUP	
10	_end	not used	STACK	STACK	DGROUP	
16		stack				
17		heap				
13		far heap				

M Memory Model

Note	Offset	Contents	Segment Name	Class	Group	Segment Register
1	0x000	PSP				
14			??_TEXT	CODE		CS
3			C_ETEXT	ENDCODE		
4		Other segments				
5			NULL	BEGDATA	DGROUP	DS,SS
6		inited data	_DATA	DATA	DGROUP	
7			CONST	CONST	DGROUP	
8	_edata	uninited data	_BSS	BSS	DGROUP	
9		common data	c_common	BSS	DGROUP	
10	_end	not used	STACK	STACK	DGROUP	
16		stack				
17		heap				
13		far heap				

C Memory Model

Note	Offset	Contents	Segment Name	Class	Group	Segment Register
1	0x000	PSP				
2			_TEXT	CODE		CS
3			C_ETEXT	ENDCODE		
4		Other segments				
5			NULL	BEGDATA	DGROUP	DS
6		inited data	_DATA	DATA	DGROUP	
7			CONST	CONST	DGROUP	
8	_edata	uninited data	_BSS	BSS	DGROUP	
9		common data	c_common	BSS	DGROUP	
10	_end	not used	STACK	STACK	DGROUP	
11		stack				SS
12		heap				
13		far heap				

L Memory Model

Note	Offset	Contents	Segment Name	Class	Group	Segment Register
1	0x000	PSP				
14			??_TEXT	CODE		CS
3			C_ETEXT	ENDCODE		
4		Other segments				
5			NULL	BEGDATA	DGROUP	DS
6		inited data	_DATA	DATA	DGROUP	
7			CONST	CONST	DGROUP	
8	_edata	uninited data	_BSS	BSS	DGROUP	
9		common data	c_common	BSS	DGROUP	
10	_end	not used	STACK	STACK	DGROUP	
11		stack				SS
12		heap				
13		far heap				

Notes:

1. Program Segment Prefix.

2. All program code is in this segment.

3. Zero length segment so we can find the end of the code.

4. Any other segments.

5. The beginning of the data segment.

6. All statically initialized data.

7. In the future compiler-generated constant data will go here.

8. Start of uninitialized data.

9. Data allocated in common blocks.

10. Used to find the end of the BSS class segments. All BSS class segments are cleared to 0 by the startup code. The STACK segment is 128 bytes long, giving DOS enough stack to fire up the program and allow the startup code to allocate the real stack. The command line is copied into the 128 bytes so it is addressable by the DS register.

11. The size of the stack is determined at run time and is allocated here. It is placed in its own segment.

12. The heap starts after the stack. Successive calls to malloc, etc., ratchet the size of the heap upwards. If an allocation blocks it from growing further, it will allocate a new segment wherever DOS will give it one. The size of the heap never shrinks.

13. Far heap data is allocated by calls to farmalloc(), etc. Each successive allocation is in its own segment, usually addressed by ES. Far heap allocations are returned to DOS with farfree().

14. The ?? is replaced by the name of the source file. The code from each module is placed in its own code segment.

15. Used to find the end of the BSS class segments. All BSS class segments are cleared to 0 by CT.OBJ. The STACK segment is of 0 length.

16. The size of the stack is determined at run time and is allocated here. It grows towards the STACK segment.

17. The heap starts after the stack. There are two cases:

    a. If (_okbigbuf != 0) then the size of the heap is 64k minus the aggregate size of the DGROUP segments and the stack.

    b. If (_okbigbuf == 0) then the size of the heap grows only as required, up to 64k minus the aggregate size of the DGROUP segments and the stack.

More information can be determined by reading C.ASM, SBRK.ASM and the linker section on how the /DOSSEG switch works. C.ASM and SBRK.ASM are on the library source disk.

General Layout

The assembly code must be divided up into code and data sections (segments). Into the code segment will go executable code and functions callable from C++ . Into the data segment go static and global data declarations.

The pseudo-ops for defining the code and data segments for each memory model are different. Therefore, use the macros begcode, endcode, begdata, enddata, that are defined in MACROS.ASM for each memory model. The general layout for an ASM source file is:

```
INCLUDE MACROS.ASM          ;define memory model macros
   (EXTRN statements for C++ functions to
   be called go here)

begdata                     ;define start of data segment
   (EXTRN statements for external data
   globals go here)

enddata                     ;define end of data segment

begcode  modulename         ;define start of code segment
   (executable code goes here)


endcode  modulename         ;define end of code segment

END                         ;define end of module
```

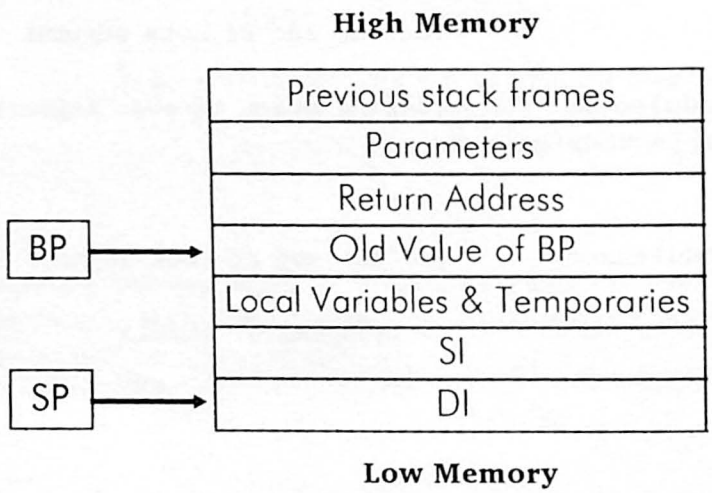
Subroutine Linkage

Since on the 8088 one cannot index off the stack pointer SP, but can off the base pointer BP, the BP register is dedicated to pointing to the current stack frame. A subroutine is called by pushing the arguments onto the stack from right to left, then the subroutine is called. The called subroutine saves the old BP on the stack, sets BP to point to it, allocates space on the stack for all local variables, and pushes SI and DI if they are needed by the function. The body of the subroutine is then executed.

The subroutine returns by popping DI and SI, deallocating space on the stack for the local variables, popping off the old value of BP, and returning. The calling code then removes the parameters from the stack.

Stack Frame

The stack frame of a function is the current state of the stack and variables in it at a point in the execution of the function. The stack frame normally looks like:



The stack grows downwards (towards lower addresses).

Function Return Values

Near pointers, ints, unsigned ints, chars, and shorts are returned in AX. Longs and unsigned longs are returned in DX,AX where DX contains the most significant 16 bits and AX contains the least significant 16 bits. Far pointers are returned in DX,AX where DX has the segment portion and AX has the offset.

Doubles are returned in AX,BX,CX,DX where AX contains the most significant 16 bits, and DX contains the least significant.

When a function returns a structure, it actually returns a pointer to the structure, which is in the static data segment (this means that functions that return structures are not re-entrant).

Register Usage

Functions may change the values in AX,BX,CX,DX, or ES. Functions must preserve the values in SI,DI,BP,SP,SS,CS,DS. The direction flag must always be set to forward.

Data Alignment

Data should be aligned along 16-bit boundaries, to maximize speed on 16-bit buses. Code is aligned along 8-bit boundaries.

Macros in MACROS.ASM

There are macros defined in MACROS.ASM which aid in the development of memory model independent assembly files. The macros are:

begcode	*;define start of code segment*
endcode	*;define end of code segment*
begdata	*;define start of initialized data segment*
enddata	*;define end of initialized data segment*
callm	*;near or far function call*
func	*;define near or far function*
c_endp	*;define end of function*
SIZEPTR	*;default pointer size in bytes (2 for T, S,M models, 4 for C,L models)*
P	*;offset of first parameter from BP*
LPTR	*;non-zero if far pointers (C or L)*
SPTR	*;non-zero if near pointers (T, S or M)*
LCODE	*;non-zero if large code (M or L)*

Simple Example for S Model

This first example calls an assembly language function to set the cursor position at x, y coordinates. All macros have been expanded and the C++ calling function has been translated to assembly language to further show how the compiler translates a C++ function. The utility used to translate the C++ function is OBJTOASM.EXE. First let's look at the C++ function:

```
main()
{
      gotoxy(10,20);
          /* set cursor position at ROW 10, COL 20 */
}
```

After compiling the C++ program to an object file, then use the utility OBJTOASM to produce the assembly language equivalent below:

```
_TEXT segment
_main:
   mov    AX,014h      ; move 20 into AX
   push   AX           ; push on stack (2 BYTES)
   mov    AX,0Ah       ; move 10 into AX
   push   AX           ; push on stack (2 BYTES)
   calln  _gotoxy      ; call gotoxy() function (PUSHES IP)
   add    SP,4         ; adjust stack pointer (4 BYTES)
   ret
_TEXT ends
```

Note that the variables are pushed on the stack from right to left.

First the column (20) is moved into AX and pushed on the stack. Next the row (10) is moved into AX and it is pushed on the stack. Finally the call to gotoxy() is made pushing the IP (instruction pointer) on the stack.

The under score appended to the function _main() and _gotoxy() was done by the compiler and conforms to Microsoft calling conventions. _TEXT segment is the CODE segment. Below is how the stack now looks after the call to _gotoxy(10,20)

Stack:

20
10
IP

Lastly, examine the assembly language function overleaf. All macros are defined in MACROS.ASM.

```
include MACROS.ASM        ;pull in defs of macros
begcode gotoxy            ;define start of code seg

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; C++ interface routine.
; Puts cursor at row,col.
; Usage:
;  void gotoxy(int row,int col);
   c_public gotoxy          ; make gotoxy global
func  gotoxy                ; define start of func
   push  BP                 ; save old stack frame
   mov   BP,SP              ; set BP to point to old BP
   mov   DH,P[BP]           ; DH = row
   mov   DL,P[BP]           ; DL = col
   mov   AH,2               ; BIOS function code to set
                              cursor pos.
   xor   BX,BX              ; page 0
   int   10h                ; BIOS video interrupt
   pop   BP                 ; restore old BP
   ret                      ;return to caller
c_endp   gotoxy             ;define end of func
endcode  gotoxy             ;define end of code seg
end
```

The assembly language function begins by first pushing BP on the stack and moving the stack pointer into BP. This will allow us to access the variables pushed on the stack by the C++ calling function.

This is done by using BP to point to offsets within the stack. In the example above, we use MOV DH,P[BP] to obtain the row number from the stack and place it in DH. The diagram below shows the variables and their positions on the stack. P expands to 4 for T, S, or C models, 6 for M and L models. It is the offset from BP to the first parameter on the stack.

Stack:

20	**BP + 6**
10	**BP + 4**
IP	**BP + 2**
previous BP	**BP + 0**

After completing the function, restore BP and return to the calling function. The 'ret' will pop IP off the stack and begin execution at the instruction following the 'calln _goto (10,20)'. The next instruction in our C++ function is 'ADD SP,4'. This will reset SP to the position it occupied befor the two 'PUSH AX' instructions. The above example pertains to T (tiny), S (small) and C (compact) models. If either L (large) or M (medium) memory models are used, CS is also pushed on the stack by the far call. This will change the position of the variables on the stack to the values below:

Stack:

20	**BP + 8**
10	**BP + 6**
CS	**BP + 4**
IP	**BP + 2**
previous BP	**BP + 0**

Using the P macro (defined in MACROS.ASM) will compensate for these differences.

Further Example

Here is an example of an assembly routine to implement the following C++ function (this is not a terribly useful function, but it illustrates how things are done). The routine is written to make it assemble correctly for any memory model:

```
                                        // C++  MODULE
extern var1;
int var2;
extern func1();
int func2(pa,a)
int *pa,a;
{
      int b;
      *pa = b;
      var2 = b + var1 + func1(&b,a);
      return a - var2;
}
```

Assembler module overleaf.

```
                                        ; Assembler MODULE
include  MACROS.ASM
begdata                             ; define start of data seg
         extrn    _var1:word
_var2    dw       0                 ; allocate var2
enddata                             ; end of data segment
if LCODE                            ; if large code model
   extrn    _func1: far             ; then far function
else
   extrn    _func1: near            ; else near function
endif
begcode     func2
   c_public func2                   ; make func2 global
func        func2                   ; define function func2
   push     BP                      ; save old frame pointer
   mov      BP,SP                   ; set new frame pointer
   sub      SP,2                    ; create room for b
   mov      AX,-2[BP]               ; AX = b
if SPTR                             ; if small memory model
   mov      BX,P[BP]                ; BX = pa
   mov      [BX],AX                 ; *pa = b
else                                ; else large memory model
   les      BX,P[BP]                ; ES:BX = pa
   mov      ES:[BX],AX              ;*pa = b
endif
   push     P+SIZEPTR[BP]           ; push a onto stack
if LPTR                             ; if far pointers
   push     SS                      ; push segment of b
endif
   lea      AX,-2[BP]               ; AX = offset of b
   push     AX
   callm    func1                   ; call func1(&b,a)
   add      SP,SIZEPTR+2            ; restore the stack
   add      AX,_var1                ; func1 returned result in
                                      AX
   add      AX,-2[BP]               ; AX = b + var1 + func1(a)
```

(Continued on following page)

(Continued . . .)

```
    mov     _var2,AX
    mov     AX,p+SIZEPTR[BP]  ;AX = a
    sub     AX,_var2          ; AX = a - var2
    mov     SP,BP             ; dump local variables
    pop     BP                ; restore old frame pointer
    ret                       ; AX has return value
c_endp      func2             ; end of function func2
endcode     func2             ; end of code segment
END                           ; end of module
```

Note that EXTRN statements for code should be outside the begcode/endcode pairs, otherwise a message about fixup errors from the linker could be generated when using the M or L models.

Running MASM

The include file MACROS.ASM sets up macros depending on which memory model is desired. The memory model is indicated by defining a symbol on the command line:

```
MASM /MX /DI8086? /R module;
```

where ? is one of S, M, C or L corresponding to the appropriate memory model. The S model is the default (you can see this by looking at the file MACROS.ASM). Do not define I8086T for T model programs, use I8086S instead. (Recall that the only difference between T and S programs is how they are linked, not how they are compiled or assembled.)

The /MX is necessary so that all global names will be case sensitive. Do not use the /ML switch, as it will cause some versions of MASM to assemble 8087 opcodes incorrectly.

The /R enables the assembling of 8087 opcodes.

Special support for MASM is built into ZTC. If a file argument to ZTC ends in .ASM, ZTC will try to assemble it with MASM. If a memory model is specified to ZTC, it will pass the appropriate define to MASM. The -g, -D, -v and -I switches to ZTC are passed through to MASM, after modifying them to be the corresponding MASM switches.

This mechanism should only be used with MASM version 5.0 or later, since the /Zd (ZTC -g) flag is not valid for earlier versions of MASM.

The file C.ASM on the library source disk (available separately) cannot be assembled with MASM versions prior to 5.0. The reason is that the older assemblers do not support the DOSSEG pseudo-op.

PART THREE

the function library

THE FUNCTION LIBRARY

Library functions are normally split into two categories

- ***ANSI C compatible functions***
- ***Non-ANSI enhanced library functions***

The ANSI C functions are those laid down in the proposed American National Standards Institute standard for the C programming language. These functions allow the greatest portability between different operating enviroments, as well as the most important MSDOS specific functions. The ANSI standard is a superset of the original language as defined by Kernighan and Ritchie and incorporates a number of the C++ enhancements into the standard C language.

The Zortech C library conforms very closely with the proposed ANSI standard, as well as containing a large number of additional functions designed to improve the programmers use of the hardware and operating system resources available to him. Functions within the library which are included in the draft ANSI standard are marked:

/* ANSI */

Use of these functions will increase the portability of applications. This section of the manual contains information about system and math error messages, include files, and functions listed in alphabetical order. Prior to this there is information about two particular groups of functions.

- ***File handling functions***
- ***Memory management functions***

Many of the Zortech functions require specialized knowledge of the 8086 processor, MSDOS internals or the PC hardware. Much of this knowledge can be obtained by reading the following -

- ***IBM PC Technical Reference (hardware)***
- ***MSDOS Programmer's Reference (from Microsoft)***

File I/O

File modes

Files can be read and written in one of two modes: text mode and binary mode. Binary is the MSDOS "native" mode. Under it, files are read and written byte-for-byte as specified. In text mode various transformations occur. These are:

Output: If a '\n' (linefeed) is specified, a '\r\n' (carriage return linefeed) is actually written.

Input: Carriage returns ('\r') are ignored on input, and a control Z (0x1A) marks the end of a file.

Ordinary text files (e.g. a source file) should use text mode, which is the default mode. Files containing binary data should use binary mode. See fopen() for details of how to open files in the different modes.

High and Low level routines

There are two sets of file I/O routines: high level and low level. The high-level routines read and write files in either text or binary mode. The low-level routines read and write files in binary mode only.

High-level and low-level routines should never be used on the same file at the same time. High-level routines should normally be used by the programmer, as they are the most portable.

Low-level File I/O

The low-level routines are the functions creat(), open(), close(), lseek(), read(), and write(). They are implemented as direct hooks into the MSDOS operating system. The file they work on is indicated by a file descriptor (known as a "handle"). I/O is unbuffered and is binary. Some special file descriptors that are always open are:

0	Standard input device
1	Standard output device
2	Standard error device
3	Standard auxiliary device
4	Standard printer device

High-level File I/O

The high-level functions including fopen(), fread(), fwrite(), and fclose() are built up from the low-level functions. They communicate by means of a stream pointer (declared as FILE *). High-level functions are buffered. Five streams are always open, they are:

stdin	Standard input
stdout	Standard output
stderr	Standard error
stdaux	Standard auxiliary
stdprn	Standard printer

To use the high-level functions, you should have the line

```
#include < stdio.h >
```

at the beginning of your source file.

Memory Allocation

HEAP

The "heap" is an area of memory available for dynamic allocation. Memory is allocated and freed using the functions malloc(), realloc(), calloc(), and free().

For S (small), M (medium), and T (tiny) models the heap space can be allocated from MSDOS all at once or just as needed. Most programs will use the default allocation scheme which fills out the data segment using all available memory space (up to 64k) during program start-up.

The second scheme of allocating heap space as needed is used for programs that execute other programs, i.e. use the spawn() functions.

Allocating memory to the heap as needed leaves the maximum amount of memory available to the child process. The second scheme is chosen by declaring and initializing a variable called _okbigbuf to a value of zero. The following example shows how the second scheme is chosen:

```
int _okbigbuf = 0;/* Use allocate as needed scheme */
```

_okbigbuf != 0 means use large disk buffers for stream I/O outside the data segment. This is all controlled by C.ASM.

C.ASM is the C startup module which is automatically linked at the start of your program and initializes the segment registers to reflect the memory model being used, as well as ensuring that the program has the correct structure as required by MS-DOS. It also sets up the memory allocation scheme which will be used to allocate memory from the data segments.

For more information on memory allocation, see the assembly language interface section of the reference manual.

Stack

The stack grows downwards toward statically allocated data. If the stack grows past its limit of space, then statically allocated data will be destroyed and the program will perform in an unexplainable manner. To check if the stack grows past the allocated limit there is a function called _chkstack().

_chkstack() can be called from functions that could use too much stack space. If this happens it will abort the program with a stack overflow message. The -s compiler switch can be used to generate stack overflow checking code on entry to each function. Although this will increase program size and affect speed, it is added as an option to help resolve stack overflow problems. If a stack overflow is detected, the message

```
STACK OVERFLOW
```

is displayed and the program will abort. Stack size can be adjusted by statically initializing the _stack global variable

```
unsigned _stack = nnnn;
```

at the beginning of your program where 'nnnn' is the size in bytes of required stack. The default stacksize is 2048 bytes. Acceptable values for nnnn are 1024 . . .30,000.

Space allocated to the stack does not take away from static data in the C and L models, since the stack is placed in its own segment.

Global Variables

There are several global variables that define the state of parts of the program. The variable names and usage are described here.

_8087

Each time a program begins execution, a determination is made as to whether an 8087 co-processor is present. If present, all floating point calculations are performed using the 8087. If not present, all floating point calculations are performed by software emulation. The _8087 global variable defined in C.ASM, can be tested to see if an 8087 is present. If the _8087 variable is equal to 0, there is no 8087 present. A value of 1 indicates that an 8087 has been found.

Future versions of the startup module will set _8087 to 3 if an 80387 is present, so your software can take advantage of the extended capabilities of the 80387. The behavior of the Zortech libraries is also controlled by this flag. For example:

```
extern int _8087;
   if(_8087)
      printf("8087 present\n");
   else
      printf("No 8087 present\n");
```

Floating Point Options

Four methods are offered to handle floating point operations:

1. Don't do any floating point. Prevent the floating point routines from being loaded in from the library.

2. Always do all floating point using the 8086. If an 8087 is present, don't use it.

3. Do floating point using the 8086. If an 8087 is present, use it for the computations instead of the 8086. For math functions in the library, switch to hand-coded 8087 routines if an 8087 is present. This option is the default.

4. Generate in-line 8087 code. An 8087 is required for the program to run.

Advantages:

1. The size of the resulting program is also significantly smaller.

2. No 8087 is necessary. Results are exactly repeatable across all 8086 machines. The compiler itself uses this option.

3. Works like 1 without an 8087, but if one is present it will be used and the program will run faster.

4. Fastest and smallest code is generated, and potentially more accurate results are obtained (the 8087 math is done using 80 bits versus 64 bits for the 8086 software floating point). The relative speed increase obtained by using the 8087 is highly dependent on the relative clock speeds of the 8087 and the regular CPU, and on what version of the 8087 you are using (80287, 80387, clones, etc.). For a PC, the improvement will be dramatic for floating-intensive programs.

Disadvantages:

1. Can't do any floating point operations.

2. Slowest code. Not as accurate. Program is largest.

3. Results may vary depending on whether or not an 8087 is present. Program is same size as in 2.

4. Requires an 8087 to operate. Executing 8087 instructions without an 8087 installed may crash the system.

Creating Option 1:

Option 1 is created by explicitly linking in the module INT.OBJ. INT.OBJ defines all the entry points in the floating point library, so none of the floating point code is pulled in from the library. If any calls are made to the floating point routines, the message

FLOATING POINT NOT LOADED

is printed by INT.OBJ and the program is halted. The same INT.OBJ is used by all memory models.

The linking in of INT.OBJ can be specified to ZTC with the [-mi] flag. ZTC will use the LIB environment variable to find INT.OBJ, and will specify it to the linker. For example:

```
ZTC -mi hello
```

Creating Option 2:

Option 2 is created by the following program stub:

```
main()
{  extern int _8087;
   _8087 = 0;                          /* never use 8087 */
   ...
}
```

Creating Option 3:

Option 3 is the default.

Creating Option 4:

Option 4 is created by compiling code with the -f flag to ZTC. For example:

```
ZTC file -f
```

Since code compiled with the -f flag will crash when run when an 8087 is not present, it is highly recommended that the following code be inserted when using the -f flag:

```
main()
{  extern int _8087;
   if (_8087 == 0)              /* if no 8087 present */
   {  printf("This program requires an 8087\n");
      exit(1);                          /* halt program */
   }
   ...
}
```

For the more adventurous programmers who are using option 4, the libraries can be modified and recompiled to remove all the non-8087 floating point code to make the program smaller, and slightly faster.

The same Zortech libraries are used with all four options.

Note that there is no change to the call/return conventions between functions, therefore obj modules that are compiled for any option can be freely linked together.

For example, an application that does a matrix inversion can have two versions of the inversion:

ZTC -c -f matrix -DINLINE8087 -omatrix87.obj
ZTC -c matrix
ZTC main matrix.obj matrix87.obj

The file matrix.c would contain:

```
#ifdef INLINE8087
void matrix87()                    /* need different name
                                        for 87 version */

#else
void matrix()
#endif
{
   ...
}
```

The file main.c would contain:

```
extern int _8087;
...
if (_8087)                    /* if there is an 8087 */
   matrix87();                   /* use 8087 version */
else
   matrix(); /* use software floating point version*/
```

This way an executable can use all the advantages of the 8087, but will still work correctly if the user doesn't have one. Similar examples can be seen in the math.c library source code.

_okbigbuf (T, S and memory models only)

This variable defines which memory allocation scheme will be used. There are two methods.

Method 1:

Allocate all available memory up to 64K to the heap upon program startup. This is the default method.

Method 2:

Allocate memory to the heap only as needed. This method is to be used if a spawn function will be needed. To use, declare _okbigbuf as shown below. For more information see "memory allocation".

Usage:

```
int _okbigbuf = 0;      /* Allocate memory as needed */
```

_okbigbuf != 0 also means use large disk buffers for stream I/O outside the data segment. This is all controlled by C.ASM.

_osmajor

This variable defines the major version number of MSDOS that is executing. The value of _osmajor would be 3 running under MSDOS 3.2 or 2 running under MSDOS 2.0.

Usage:

```
extern unsigned char _osmajor;
```

_osminor

This variable defines the minor version number of MSDOS that is executing. The value of _osminor would be 11 running under MSDOS 2.11 or a 1 running under MSDOS 3.1.

Usage:

```
extern unsigned char _osminor;
```

_psp

This variable contains the segment paragraph address of the program segment prefix. This can be used to construct a far pointer allowing access to the program segment prefix from within the program.

Usage:

```
extern unsigned _psp;
```

errno

This variable will be assigned error numbers after using some library functions, if errors occur. Values of errno are:

E2BIG 100
EACCES 5
EBADF 6
EDOM 102
EEXIST 80
EMFILE 4
ENOENT 2
ENOEXEC 01
ENOMEM 8
ENOTDIR 3
ERANGE 103

Usage:

```
extern int errno;
```

See also errno.h for a description of each error number.

ERROR MESSAGES

The tables below list error messages returned by the perror() function with the associated value in errno.

E2BIG	Argument list too long. (Argument list exceeds 128 bytes.)
EACCES	Permission denied. (E.g. writing to a file opened for read only.)
EBADF	Bad file descriptor.
EDOM	Math argument out of domain.
EEXIST	File exists.
EMFILE	Too many open files.
ENOENT	No such file or directory.
ENOEXEC	Exec format error.
ENOMEM	Not enough core. (E.g. not enough memory for exec function.)
ENOTDIR	Path not found.
ERANGE	Result out of range. (An argument to a math function is too large.)

See the library example for the perror() function for a description of the use of these error messages.

Math Errors

Errors occurring during the execution of math functions are returned by the function matherror().

math.h

DOMAIN	Invalid operation.
OVERFLOW	Result to large to be represented.
PLOSS	Partial loss of precision.
SING	Argument singularity, (E.g. a zero value passed to log(); a function that requires non-zero).
TLOSS	Total loss of precision.
UNDERFLOW	Result too small to be represented.

HEADER FILES

assert.h

Defines the assert() macro.

If the identifier NDEBUG has not been defined on the compiler command line or through a #define directive, the macro is defined to test a given expression. If the assertion is false, a message is printed and the program terminates.

If NDEBUG is defined, assert is defined as empty text. The effect is to comment out all assert statements.

conio.h

Declares prototypes for console and I/O functions.

getch *getche* *inp* *inpw* *outp* *oupw*

ctype.h

Defines macros and constants used in character classification.

isalnum	*isalpha*	*isascii*	*iscntrl*	*isdigit*
isgraph	*islower*	*isprint*	*ispunct*	*isspace*
isupper	*isxdigit*	*toascii*		

Also included in ctype.h is a global declaration of the array _ctype. The _ctype array is a table of ASCII based character classification codes.

direct.h

Declares prototypes for the directory handling functions.

chdir *getcwd* *mkdir* *rmdir*

disp.h

Defines macros, manifest constants and prototypes for the direct screen access functions.

disp_close	*disp_eeol*	*disp_eeop*
disp_endstand	*disp_flush*	*disp_getmode*
disp_move	*disp_open*	*disp_pokew*
disp_printf	*disp_putc*	*disp_reset43*
disp_scroll	*disp_set43*	*disp_setattr*
disp_setmode	*disp_usebios*	*disp_startstand*
disp_setcursortype		

dos.h

Defines macros, manifest constants, MS-DOS and IBM PC interface function declarations. These include I/O port access, file mode definitions, interrupt service routines, and register structure types.

Function prototype declarations for the dos- package.

dos_abs_disk_read *dos_abs_disk_write*
dos_getdiskfreespace *dos_get_ctrl_break*
dos_get_verify *dos_set_ctrl_break*
dos_set_verify

Function prototype declarations for MS DOS directory functions.

findfirst *findnext*

Function prototype declarations for far memory allocation.

farmalloc *farcalloc* *farrealloc*
farfree *farcoreleft*

Function declarations for I/O port access.

inp *inpw* *outp* *outpw*

Register structure types used by intdos(), intdosx(), int86(), int86x().

WORDREGS *BYTEREGS* *REGS* *SREGS*

Macros to manipulate far pointers (segment:offset):

FP_SEG *FP_OFF* *MK_FP*

fg.h

Defines manifest constants, types, and function prototypes for flash graphics package. The following correspond to graphics modes:

FG_CGAMEDRES	0x04	IBM CGA in 320*200 color (mode 0x04)
FG_CGAHIRES	0x06	IBM CGA in 640*200 (mode 0x06)
FG_EGACOLOR	0x0E	IBM EGA with regular color display (mode 0x0E)
FG_EGAMONO	0x0F	IBM EGA with IBM monochrome monitor (mode 0x0F)
FG_EGAECD	0x10	IBM EGA with enhanced color display (mode 0x10)
FG_VGA11	0x11	IBM VGA in 640*480, 1 color
FG_VGA12	0x12	IBM VGA in 640*480, 16 colors
FG_VGA13	0x13	IBM VGA in 320*200, 256 colors
FG_HERCHALF	0x02	Hercules monochrome, 1 display page
FG_HERCFULL	0x03	Hercules monochrome, 2 display pages
FG_TOSHIBA	0xFA	Toshiba J-3100

Functions:

fg_adjustxy	*fg_drawarc*	*fg_writebox*
fg_drawbox	*fg_drawdot*	*fg_drawellipse*
fg_drawline	*fg_drawlinep*	*fg_drawmatrix*
fg_drawthickline	*fg_enclose*	*fg_fillbox*
fg_flush	*fg_getcolormap*	*fg_init*
fg_init_all	*fg_init_cga*	*fg_init_cgamedres*
fg_init_egacolor	*fg_init_egaecd*	*fg_init_egamono*
fg_init_herc	*fg_blit*	*fg_init_toshiba*
fg_init_vga11	*fg_init_vga12*	*fg_init_vga13*
fg_putc	*fg_puts*	*fg_readbox*
fg_readdot	*fg_restore*	*fg_save*
fg_setactivepage	*fg_setcolormap*	*fg_setdisplaypage*
fg_text_mode	*fg_term*	*fg_setlinepattern*

Line types:

FG_LINE_MEDIUM_DOTTED	*FG_LINE_LONG_DASH*
FG_LINE_DENSE_DOTTED	*FG_LINE_SHORT_DASH*
FG_LINE_MEDIUM_DASHED	*FG_LINE_MAX*
FG_LINE_DASH_DOTTED	*FG_LINE_SOLID*
FG_LINE_SPARSE_DOTTED	*FG_LINE_USER_DEFINED*
FG_LINE_DASH_W_2_DOTS	

Colors:

FG_BLACK	*FG_BLUE*	*FG_GREEN*
FG_CYAN	*FG_BLUE_GREEN*	*FG_RED*
FG_PURPLE	*FG_YELLOW*	*FG_WHITE*
FG_LIGHT_BLUE_GREEN		*FG_GRAY*
FG_LIGHT_BLUE	*FG_LIGHT_GREEN*	*FG_MAGENTA*
FG_LIGHT_CYAN	*FG_LIGHT_RED*	*FG_BROWN*
FG_LIGHT_MAGENTA		*FG_BLINK*
FG_HIGHLIGHT	*FG_LIGHT_WHITE*	

errno.h

Defines constant mnemonics used by perror.

See the section on error messages.

int.h

Defines function declarations and structure for interrupt service routines.

int_getvector	*int_intercept*	*int_off*
int_restore	*int_setvector*	*int_on*

io.h

Defines function declarations for low level file handling.

close	*creat*	*delete*	*filesize*	*getDS*
isatty	*lseek*	*open*	*read*	*remov*
rename	*write*	*unlink*		

limits.h

Defines limits and ranges of numerical quantities

math.h

Declares prototypes for the math functions.

acos	*asin*	*atan*	*atan2*	*atof*
ceil	*cos*	*cosh*	*exp*	*fabs*
floor	*fmod*	*frexp*	*hypot*	*ldex*
log	*log10*	*modf*	*poly*	*pow*
sin	*sinh*	*sqrt*	*tan*	*tanh*

msmouse.h

Declares prototypes for the MS-MOUSE functions.

msm_condoff	*msm_getpress*	*msm_getrelease*
msm_getstatus	*msm_hidecursor*	*msm_init*
msm_lightpenoff	*msm_lightpenon*	*msm_readcounters*
msm_setareax	*msm_setareay*	*msm_setcurpos*
msm_setgraphcur	*msm_setratio*	*msm_settextcur*
msm_showcursor	*msm_signal*	*msm_setthreshhold*
msm_term		

process.h

Declares prototypes for all process control functions.

_exit	*execl*	*execlp*	*execv*	*execv*
spawnl	*spawnlp*	*spawnv*	*spawnvp*	

setjmp.h

Declares type jmp_buf used by setjmp and longjmp (also declared).

signal.h

Defines signal values as below.

SIG_DFL	*SIG_ERR*	*SIG_IGN*	*SIGABRT*	*SIGFPE*
SIGILL	*SIGINT*	*SIGSEGV*	*SIGTERM*	

Declares raise() and signal() functions

sound.h

Declares prototypes for the sound functions.

sound_beep	*sound_click*	*sound_tone*

stdarg.h

Defines macros used for reading variable length argument lists.

va_list *va_start* *va_arg* *va_end*

stddef.h

Defines common data types.

errno	Global variable containing error message no.
ptrdiff_t	Synonym for type that is the difference between 2 ptrs.
size_t	Synonym for type that can be used to hold the size of a variable.

stdio.h

Defines constants, macros, types, and function prototypes for stream I/O.

bdos	*clearerr*	*fclose*	*feof*	*ferror*
fflush	*fgetc*	*fgets*	*fileno*	*filesize*
flushall	*fopen*	*fprintf*	*fputc*	*fputs*
fread	*freopen*	*fscanf*	*fseek*	*ftell*
fwrite	*getc*	*getch*	*getchar*	*getche*
gets	*max*	*min*	*printf*	*putc*
putchar	*puts*	*rewind*	*stdaux*	*stderr*
stdin	*stdout*	*stdprn*	*scanf*	*setbuf*
setvbuf	*sprintf*	*sscanf*	*ungetc*	*vfprintf*
vprintf	*vsprintf*			

BUFSIZ	Default buffer size used in stream I/O.
_NFILE	Defines the no. of open files allowed at once
EOF	Defined value to be the returned value by an I/O routine upon end of file.
NULL	Defined NULL pointer value.
BIGBUF	Buffer size to use in stream I/O if _okbigbuf = 1

stdlib.h

Declares prototypes for the following functions:

abort	*abs*	*atexit*	*atof*	*atoi*
bsearch	*calloc*	*div*	*exit*	*free*
getenv	*itoa*	*labs*	*atol*	*ldiv*
malloc	*qsort*	*rand*	*realloc*	*srand*
strtod	*strtol*	*strtoul*	*system*	

string.h

Declares prototypes for string functions.

strcat	*strchr*	*strcmp*	*strcmpl*	*strcpy*
strcspn	*strdup*	*strerror*	*strlen*	*strlwr*
strncat	*strncmp*	*strncpy*	*strnset*	*strpbrk*
strrchr	*strrev*	*strset*	*strspn*	*strstr*
strtok	*strupr*	*swab*		

Declares prototypes for memory functions.

memcpy	*memcmp*	*memchr*	*memset*

sys\stat.h

Defines flags used for opening and creating files. Declares function prototypes for stat() and fstat().

S_IFCHR	*S_IFDIR*	*S_IFMT*	*S_IFREG*
S_IEXEC	*S_IREAD*	*S_IWRITE*	

stat.h cannot be included in C++ files, stat.h is for C files only because of the name conflict between struct stat and int stat().

time.h

Declares prototypes for time functions.

asctime	*clock*	*ctime*	*difftime*	*gmtime*
localtime	*mktime*	*strftime*	*time*	

LIBRARY FUNCTIONS

This section describes the library functions in alphabetical order. Each function has a number of sub-headings:

Usage
Description
Example
Return Value
See Also

Here is a description of each sub-heading.

function name

Each library function begins with the function name. Note that each function name appears in the proper case to be used in a C program, including any leading underscore '_', upper-case 'FP_OFF', and mixed case 'getDS'. It is important to observe the case used for a function because 'printf' and 'PRINTF' are not equivalent.

Usage:

First is the compiler directive #include < filename.h >. < filename.h > is the name of the header file containing the prototype of the function. The prototype declares the data type for return values and arguments passed to a function. Header files can also contain types, structures, and constant definitions to be used by a function. Header files are typically included at the beginning of the program prior to the first function.

Following the directive is the function prototype. It appears the same as defined in the header file. It is possible to declare a function by typing in the function prototype instead of including the header file.

Because a function may depend on other data types defined in the header file, proper error checking is assured by including the header file rather than just declaring the function prototype.

Description:

Encapsulates what the function does. It describes what arguments are to be passed to the function and any other details needed to use the function.

Example:

In almost all cases a complete stand alone executable example is presented which you may type in and execute. Each example includes all header files needed by the functions used. You will note that in almost all cases the stdio.h header file is included. This is because printf is a function which accepts a variable number of arguments, so to assure proper error checking, the header file which prototypes printf (stdio.h) is included.

Return Value:

Explains any values returned by a function. In many cases the return value indicates success or failure. Although it is not always necessary to check the return values; checking for errors will help to insure proper execution of your program.

See Also:

Lists other related or complementary functions.

abort

Usage:

#include <stdlib.h>
void abort(void); **/* ANSI */**

Description:

abort terminates the currently executing program. Same as a function call to _exit with a non-zero status.

Example:

```
#include <stdlib.h>
#include <stdio.h>
main()
{
   FILE *fp;
   fp = fopen("file.dat","r");
   if(fp == NULL)
      {
         fprintf(stderr,"Could not open file.dat\n");
         abort();
      }
   printf("File opened\n");
}
```

Return Value:

There is no return value.

See Also:

exit, _exit, raise

abs

Usage:

#include <stdlib.h>
int abs(int i); **/* ANSI */**

Description:

abs produces the absolute value of its integer argument.

Example:

```
#include <stdlib.h>
#include <stdio.h>
main()
{
   int numbr, absval;
   numbr = -3;
   absval = abs(numbr);
   printf("The abs of (%d) is %d\n",numbr,absval);
}
```

Return Value:

abs returns the absolute value of its integer argument.

See Also:

labs

acos

Usage:

#include <math.h>
double acos(double x); **/* ANSI */**

Description:

acos returns the arc cosine of x in the range of 0 <=acos<=pi. The argument x must be in the range of -1<=x<=1.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
    double x, y;
    x = .94;
    y = acos(x);
    printf("The arccosine of %f is %f\n",x,y);
}
```

Return Value:

acos returns the arc cosine of the input value. If the value x is less than -1 or greater than 1, acos sets errno to EDOM, prints a DOMAIN error message to stderr and returns 0. Error handling can be modified through the function matherr.

See Also:

asin, atan, atan2

asctime

Usage:

#include<time.h>
*char *asctime(struct tm *ntime);* **/* ANSI */**

Description:

asctime converts a time structure into an ASCII string of 26 characters including the 0 having the form of:

DDD MMM dd hh:mm:ss YYYY\n\0

Where:

DDD	=	day of the week
MMM	=	month
dd	=	day of the month
hh:mm:ss	=	hour:minutes:seconds
YYYY	=	year

The time value ntime can be obtained by a call to localtime() which returns a pointer to the tm structure defined in time.h.

Example:

```
#include <time.h>
#include <stdio.h>
main()
{
    timet tclock;
    time(&tclock);
    printf("The current date and time: %s\n",
          asctime(localtime(&tclock)));
}
```

Return Value:

Returns a pointer to a character string containing the date and time. The string is static and is overwritten with each call to asctime.

See Also:

clock, ctime, difftime, localtime

asin

Usage:

#include <math.h>
double asin(double x); **/* ANSI */**

Description:

asin returns the arc sine of x in the range of -pi/2 <=asin<= pi/2. The value of x must be between -1<=x<=1.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
   double x, y;
   x = .94;
   y = asin(x);
   printf("The arcsine of %f is %f\n",x,y);
}
```

Return Value:

asin returns the arc sine of the input value. If the value is less than -1 or greater than 1, asin sets errno to EDOM, prints a DOMAIN error message to stderr and returns 0. Error handling can be modified through the function matherr.

See Also:

atan, atan2, acos

assert

Usage:

#include <assert.h>
void assert(expression); **/* ANSI */**

Description:

The assert macro is useful for adding internal consistency checks to programs. When the assert macro is executed it checks the expression argument and if it is zero, then the following message is printed on the standard error device and the program aborts.

Assertion failure: 'expression' on line ?? in file '???'

Example:

```
#include <assert.h>
#include <stdio.h>
char *string = "";                    /* empty string */
int value = 1;
main()
{
   assert(value  >0);
   printf("Passed assert(value  >0)\n");
   assert(*string != '\0');
}
```

Return Value:

There is no return value.

atan, atan2

Usage:

#include <math.h>
double atan(double x); **/* ANSI */**
double atan2(double y, double x); **/* ANSI */**

Description:

atan returns the arc tangent of x in the range of -pi/2 to pi/2; atan2 returns the arc tangent of y/x in the range of -pi to pi. atan2 uses the signs of y and x to determine the quadrant of the return value.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
   double x, y;
   x = 1.0;
   y = atan(x);
   printf("The arctangent of %.7f is %.7f\n",x,y);
}
```

Return Value:

Both atan and atan2 return the arc tangent of the input value(s). If x and y are both 0, atan2 sets errno to EDOM, prints a DOMAIN error message to stderr, and returns 0. Error handling can be modified through the function matherr.

See Also:

acos, asin

atof, atoi, atol

Usage:

#include <stdlib.h>
*double atof(char *nptr);* **/* ANSI */**
*int atoi(char *nptr);* **/* ANSI */**
*long atol(char *nptr);* **/* ANSI */**

Description:

These convert the string pointed to by nptr into a double (atof), integer (atoi), or long (atol). The string may have leading spaces, tabs, and + or -. Conversion stops on the first unrecognized character. If there are no recognized characters, the result is 0.

Example:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
   printf("atof function = %e\n",atof("123.5e15"));
   printf("atoi function = %d\n",atoi("153"));
   printf("atol function = %ld\n",atol("1234 String"));
}
```

Return Value:

Each function returns the double, int, or long value derived from converting the string. Zero is returned if the input string does not have any recognizable characters.

See Also:

ecvt, fcvt, strtol, strtod, scanf

bdos, bdosx

Usage:

#include <dos.h>
int bdos(int func, unsigned DX, unsigned AL);
*int bdosx(int func, char *ptr);*

Description:

Calls an MSDOS function (int 21h). func is the function number and DX and AL are the values to be loaded into the DX and AL registers prior to the function call. The bdosx call is the same as bdos except the ptr argument is loaded into the DS:DX register pair.

Example:

```
#include <dos.h>
main()
{
   bdos(2,'C')                /* display the letter C */
}
```

Return Value:

Both functions return the value in AX after the system call.

See Also:

intdos, intdosx, int86, int86x

bioskey
_bios_keybrd

Usage:

#include <bios.h>
int bioskey(int flag);
int _bios_keybrd(int flag);

Description:

These names refer to the same function. The two spellings are for compatibility with other compilers. bioskey passes flag to the BIOS interrupt 0x14 - the keyboard interrupt. The values for flag are:

0	Read the next key value from the keyboard input buffer. Wait for one if there isn't any available.
1	Determine if any keys are in the keyboard input buffer.
2	Read the status of the shift keys.

Return Value:

If flag is 0, the value returned is the key value. The ASCII value of the key is returned in the low byte, and the scan code in the high byte. If the low byte is 0, then the key value is not an ASCII key (it might be an arrow or function key). If flag is 1, 0 is returned if there are no keys in the input buffer, otherwise the next key value is returned. The key value is not removed from the input buffer, and is still available to be read. If the flag is 2, the value returned is the state of the shift keys encoded into the bits as follows:

0x01	Right shift key is down
0x02	Left shift key is down
0x04	Ctrl key is down
0x08	Alt key is down
0x10	Scroll Lock is toggled
0x20	Num Lock is toggled
0x40	Caps Lock is toggled
0x80	Ins is toggled

Example:

```
#include <bios.h>
#include <stdio.h>
main()
{
   int key,shift;
   int lastshift = 0;
   while (1)
   {
      shift = bioskey(2);
                          /* If shift status changes */
      if (shift != lastshift)
         printf("shift = 0x%02x\n",shift);
                             /* If a key is available */
      if (bioskey(1))
      {                                  /* Read the key */
         key = bioskey(0);
         if ((key & 0xFF) == 'q')
            break;
         printf("key = 0x%04x\n",key);
      }
      lastshift = shift;
   }
}
```

bsearch

Usage:

```
#include <stdlib.h>
void *bsearch(const void *key,const void *base,size_t num,
size_t width,int (*cmp)(const void *elem1,const void *elem2));
                                                    /* ANSI */
```

Description:

bsearch performs a binary search of a sorted array of num elements, which is pointed to by base, for an element which matches key. Each item of the array is width bytes. The function used in the search is *cmp. This function must be supplied by the programmer. The cmp function takes two arguments as pointers to the element in the array. The *cmp function must return one of the following values:

<0	elem1 is less than elem2
0	elem1 and elem2 match
0	elem1 is greater then elem2

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE(arr) (sizeof(arr) / sizeof(arr[0]))
int array[] = {1254,3427,1111,3901,6677,0101};
int intcmp(int *p1, int *p2)
{
     return(*p1 - *p2);
}
main()
{
   int *pointer;
   int key = 3901;
   pointer = (int *) bsearch(&key,
              array,SIZE(array), sizeof(int),intcmp);
   if(pointer)
   {
      printf("[%d] is in array\n",key);
   }
   else
   {
      printf("[%d] is not in array\n",key);
   }
}
```

Return Value:

Returns a pointer to the matching elem, or NULL if not found.

calloc

Usage:

#include <stdlib.h>
*void *calloc(unsigned numelems, size_t sizelem);*

/* ANSI */

Description:

Allocates a block of memory that is numelems * sizelem bytes in size. The memory is cleared and a pointer to it is returned. If there is an error (e.g. insufficient memory), NULL is returned. If either numelems or sizelem is 0, NULL is returned. The block of memory requested must be less than 64K.

Return Value:

A pointer to the allocated memory is returned on success, otherwise a NULL pointer is returned.

See Also:

free, malloc, realloc, farcalloc

Example:

```
                    /* compile with C or L model */

#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#define num 50
main()
{
   long *buffer;
   unsigned int segment,offset;
   buffer = calloc(num,sizeof(long));
   if(!buffer)
      {
         fprintf(stderr,"Calloc failed\n");
         abort();
      }
   segment = FP_SEG(buffer);
   offset = FP_OFF(buffer);
   printf(
      "Memory allocated at segment: %x offset: %x\n",
                                        segment,offset);
   free(buffer);
}
```

ceil

Usage:

#include <math.h>
double ceil(double x); **/* ANSI */**

Description:

ceil returns a double value representing the smallest integer that is greater than or equal to x.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
    double x = 5.3,result;
    result = ceil(x);
    printf("The ceil of %f is %f\n",x,result);
}
```

Return Value:

Returns the double value of the smallest integer not less than x.

See Also:

floor, fmod

chdir

Usage:

#include <direct.h>
*int chdir(char *path);*

Description:

chdir is a system call that changes the current directory to the directory specified in the path argument. Path must be a valid directory.

Example:

```
#include <direct.h>
#include <stdio.h>
char *path = "\\temp";
main()
{
   int result;
   result = chdir(path);
   if(result != 0)
      printf("Could not change to directory %s\n",path);
   else
      printf("Directory changed to %s\n",path);
}
```

Return Value:

Returns a 0 upon success otherwise a -1 and errno is set.

See Also:

mkdir, rmdir

_chkstack

Usage:

#include <stdef.h>
size_t _chkstack(void);

Description:

_chkstack determines if the stack has grown larger than the memory allocated for it and if so, aborts the program with a stack overflow message. This function should be called in recursive functions or other functions that might use a lot of stack space. Stack checking can be performed automatically with the -s switch to ZTC2.

Example:

```
                    /* do_tree is a recursive func */
do_tree(t)
struct tree *t;
{
#ifdef __ZTC__          /* defined if Zortech C \ C++ */
_chkstack();
#endif
    .
    .
    do_tree(t->left);
    do_tree(t->right);
}
```

Return Value:

Returns the number of bytes left on the stack, if the stack hasn't overflowed.

clearerr

Usage:

#include <stdio.h>
*void clearerr(FILE *fp);* **/* ANSI */**

Description:

Clears error and EOF flags associated with stream fp.

Example:

```
#include <stdio.h>
main()
{
   FILE *stream;
   char *string = "Sample data";

   stream = fopen("file.dat","r");
   fprintf(stream,"%s\n",string);
   if(ferror(stream))
      {
         fprintf(stderr,"Write error to file.dat\n");
         clearerr(stream);
      }
   printf("No error writing to file\n");
   fclose(stream);
}
```

Return Value:

none

See Also:

feof, ferror

clock

Usage:

#include <time.h>
clock_t clock(void); **/* ANSI */**

Description:

Determines the processor time used by the calling process at the time clock is executed. IBM PCs have granularities of 1/18 sec. Other MS DOS machines have only 1 sec granularity, although the time is always returned in 1/100ths of a second.

Example:

```
#include <stdio.h>
#include <time.h>
main()
{
   clock_t process;
   process = clock()/CLK_TCK;
   printf("Processor time in seconds %ld",process);
}
```

Return Value:

Returns an approximation of the processor time used thus far by the calling program. Division of the return value of clock by the value of the CLK_TCK macro yields the time in seconds.

See Also:

asctime, ctime, difftime, localtime

close

Usage:

#include <io.h>
int close(int fd);

Description:

Closes file associated with file descriptor handle fd.

Example:

```
#include <io.h>
#include <dos.h>
#include <stdio.h>
main()
{
int fd,result;
   fd = open("CON",O_RDONLY);
   if(fd < 0 )
      {
         fprintf(stderr,"Can't open CON\n");
         abort();
      }
   result = close(fd);
   printf("Result code from close %d\n",result);
}
```

Return Value:

Close returns 0 if successful, or -1 and errno is set.

See Also:

open, unlink

cos, cosh

Usage:

#include <math.h>
double cos(double x); **/* ANSI */**
double cosh(double x); **/* ANSI */**

Description:

cos and cosh return the cosine and hyperbolic cosine of x, respectively. For cos, x is in radians.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
   double value = 2.5556,result,hycos;
   result = cos(value);
   printf("The cosine of %f is %f\n",value,result);
   hycos = cosh(value);
   printf(
   "The hyperbolic cosine of %f is %f\n",value,hycos);
}
```

Return Value:

cos returns the cosine of x. cosh returns the hyperbolic cosine of x. Zero is returned on error.

creat

Usage:

#include <io.h>
#include <sys\stat.h>
*int creat(char *name, int pmode);*

Description:

The system call creat() either creates a new file or opens and truncates an existing file. The permission setting, pmode, sets the new file's reading and writing permission after the file is closed for the first time. pmode takes one or both of the following values defined in stat.h:

```
#define S_IREAD 0000400  /* read permission, owner */
#define S_IWRITE 0000200/* write permission, owner */
```

When both constants are required, they should be joined with the bitwise OR operator. Under MS DOS it is not possible to give write only permission. pmode is currently ignored.

Example:

```
#include <io.h>
#include <sys\stat.h>
main()
{
   int fp;
   fp = creat ("file.dat", S_IREAD | S_IWRITE);
   if (fp == -1)
      printf("Cannot creat file.dat\n");
}
```

Return Value:

creat returns a -1 if an error occurred and sets errno to one of the following:

ENOENT	No such file or directory.
EMFILE	Too many open files.
EACCES	Permission denied.

Similar to chdir(), close(), etc.

creat returns an MSDOS file handle if the file was created.

See Also:

dos_creat, close, fopen, open

ctime

Usage:

#include <time.h>
*char *ctime(time_t *ntime);* **/* ANSI */**

Description:

Converts the calender time (type time_t) pointed to by ntime to time in the form of an ASCII string.

Example:

```
#include <time.h>
#include <stdio.h>
time_t ntime;
main()
{
   time(&ntime);
   printf("The current time is %s\n",ctime(&ntime));
}
```

Return Value:

Returns pointer to a static ASCII string of 26 characters of the form:
DDD MMM dd hh:mm:ss YYYY\n\0

Where:

DDD	=	day of the week
MMM	=	month
dd	=	day of the month
hh:mm:ss	=	hour:minutes:seconds
YYY	=	the year

The string is overwritten by each call to ctime().

See Also: asctime, clock, difftime, localtime, time

difftime

Usage:

#include <time.h>
double difftime (time_t time2, time_t time1);

/* ANSI */

Description:

difftime subtracts time1 from time2.

Example:

```
#include <stdio.h>
#include <time.h>
main()
{
   register int delay;
   time_t start, finish;
   time(&start);
   for (delay = 0; delay < 1000; delay++)
      printf("%d \n", delay);
   time (&finish);
   printf("The delay lasted %.0f seconds",
      difftime(start, finish));
}
```

Return Value:

Returns the difference between time1 and time2 in seconds.

See Also:

asctime, clock, ctime, localtime, time

Display Package

Usage:

#include < disp.h >

void disp_close(void);
Flushes output and terminates use of the display package.

void disp_box(int type, int attr, unsigned row, unsigned col, unsigned brow, unsigned rcol);
Draws a box of border *type* with attribute *attr* starting at top left *row, col* and finishing at bottom right *brow, rcol.*

Values for type are:

0 = Double Line Border
1 = Single Line Border
2 = Solid Border

void disp_eeol(void);
Erases to the end of the current line, including the character under the cursor.

void disp_eeop(void);
Erases from the current cursor position to the end of the screen.

void disp_endstand(void);
Ends the reverse video mode. This should be called after disp_startstand.

void disp_flush(void);
Flushes output. This is necessary when the screen must be up-to-date, such as when input is requested from the user.

int disp_getmode(void);
Returns the current video mode. For mode types see disp_setmode.

void disp_move(int row,int col);
Moves the cursor to the row and column specified. The home position is 0,0 which is the upper left corner of the display.

void disp_open(void);
Initializes the display package. The type of display and the screen memory address are determined during this call. All globals are initialized.

unsigned disp_peekw(int row,int col);
Reads attribute/character from video display.

*unsigned disp_peekbox(unsigned *save, unsigned row, unsigned col, unsigned brow, unsigned rcol)*
Stores a rectangular area of the screen defined by top left *row, col* and bottom right *brow, rcol* in the user supplied buffer *save.*

*void disp_pokebox(unsigned *save, unsigned row, unsigned col, unsigned brow, unsigned rcol)*
Restores a rectangular area of the screen defined by top left *row, col* and bottom right *brow, rcol* from the buffer *save.*

void disp_pokew(int row,int col,int attr char);
Pokes an attribute and character into a specified row and column.

*void disp_printf(char *format,...);*
The formatted print routine for writing directly to the screen. For more information see the printf function description.

void disp_putc(int c);
Writes the character c to the current screen location using the character attribute set by disp_setattr. Handles tabs, carriage returns and backspaces, etc correctly.

void disp_reset43(void);
Resets display from 80 by 43 EGA text mode back to 80 by 25 mode.

void disp_scroll(int lines,unsigned ulrow,unsigned ulcol,
unsigned lrrow,unsigned lrcol,unsigned attr);
Scrolls the screen up:

lines: number of lines to scroll
ulrow: upper left row of display area
ulcol: upper left column of display area
lrrow: lower right row of display area
lrcol: lower right column of display area
attr: video attribute to use on uncovered areas.

void disp_set43(void);
Sets display mode to 80 by 43 EGA text mode. Should not be used while display package is open. Must have an EGA or VGA for this to work. The actual resulting # of lines is written in disp_numrows.

void disp_setattr(int attr);
Sets attribute of chars written by disp_putc and disp_printf. Below is a list of the attributes found in the disp.h file.

DISP_REVERSEVIDEO
DISP_NORMAL
DISP_UNDERLINE
DISP_NONDISPLAY

The following attribute bits can be OR'ed in:

DISP_INTENSITY
DISP_BLINK

*void disp_setcursortype(int startline*256 + endline);*
Sets the cursor type (size). The DOS default cursor types are:

	Start	End	Use
Monochrome	11	12	((11*256)+12)
Color	6	7	((6*256)+7)

void disp_setmode(unsigned char mode);
Sets the video mode. If the current video mode is not required, a call to this function **must** be made before the call to disp_open. See below for modes.

0	- 40 x 25 B/W Alpha	- color display card/EGA
1	- 40 x 25 color Alpha	- color display card/EGA
2	- 80 x 25 B/W Alpha	- color display card/EGA
3	- 80 x 25 color Alpha	- color display card/EGA
4	- 320 x 200 color graphs	- color display card/EGA
5	- 320 x 200 B/W graphs	- color display card/EGA
6	- 640 x 200 B/W graphs	- color display card/EGA
7	- 80 x 25 B/W Alpha	- Mono. display card / EGA with mono display.
D	- 320 x 200 16 color	- EGA
E	- 640 x 200 16 color	- EGA
F	- 640 x 350 4 color	- EGA with mono display
10	- 640 x 350 16 color	- EGA

void disp_startstand(void);
Causes all characters printed after this call to be displayed in reverse video mode. The reverse video mode is turned off using the disp_endstand function call.

void disp_usebios(void);
Sets up the display package to write chars to the screen using the BIOS, instead of writing directly to the display. This increases portability while reducing speed. This should be called after disp_open.

Description:

The display package provides fast screen I/O for the PC and true compatibles. These routines write directly into screen memory thereby gaining their speed. These routines will NOT WORK on machines that do not have an IBM-PC compatible display or BIOS. All the globals below are initialized when the display package is opened with a call to disp_open(). The current status of any global can be tested in your program. These globals are to be READ ONLY.

unsigned char disp_mono	0 if color and 1 if monochrome.
unsigned char disp_snowycga	!=0 if snowy IBM CGA.
unsigned char disp_mode	Current display mode. See disp_setmode().
unsigned char disp_inited	!=0 if display package open.
unsigned char disp_ega	!=0 if IBM EGA or VGA
unsigned disp_base	Segment of video display RAM (MDA=0B000) (CGA=0B800)
int disp_numrows	No. of display rows.
int disp_numcols	No. of display columns.
int disp_cursorrow	Row of cursor position.
int disp_cursorcol	Col. of cursor position.

Return Value:

disp_getmode returns the current video mode.

See Also:

printf, putc

Example:

```
#include < disp.h >
#include < conio.h >
char *string = "Blinking Example String";
int cursor = ((6*256)+11);
main()
{
char ch;
   disp_open();
   disp_startstand();
   disp_printf(  "String displayed after call to "
                 "disp_startstand\n");
   disp_endstand();
   disp_setattr(DISP_BLINK | DISP_NORMAL);
   disp_printf("%s\n",string);
   disp_setattr(DISP_INTENSITY | DISP_NORMAL);
   disp_printf("Attribute set to High Intensity\n");
   disp_setattr(DISP_NORMAL);
   disp_usebios();
   disp_printf("String displayed using BIOS\n");
   disp_printf("Change cursor size, Press any key: ");
   disp_setcursortype(cursor);
   disp_flush();
   ch = getch();
   disp_close();
}
```

div

Usage:

#include <stdlib.h>
div_t div(int numerator, int denominator);

/* ANSI */

Description:

div divides the numerator by denominator, returning the quotient and the remainder.

Example:

```
#include <stdlib.h>
main()
{
   div_t answ;
   int ina,inb;
   puts("Enter two integers:");
   scanf("%d %d",&ina,&inb);
   answ = div(ina,inb);
   printf( "The quotient is %d, and the remainder"
           "is %d\n", answ.quot,answ.rem);
}
```

Return Value:

The div function returns a structure type div_t.

See Also:

ldiv

DOS Package

Usage:

```
#include <dos.h>
int dos_abs_disk_read (int drive,int num_sec,
int start_sec,char *buffer);
int dos_abs_disk_write(int drive,int num_sec,
int start_sec,char *buffer);
int dos_creat( char *name,unsigned attribute);
int dos_get_cntl_break(void);
long dos_getdiskfreespace(int drive);
int dos_get_verify(void);
void dos_set_verify(int on_off);
void dos_set_cntl_break(int on_off);
```

Description:

dos_abs_disk_read, dos_abs_disk_write

Transfer control directly to BIOS to perform the disk read or write. The drive is a 0 for A:, 1 for B: and so on up to 25. The number of sectors to read or write is specified in num_sec. Start_sec defines the first sector for operation. Buffer is the source or destination memory address for the operation.

dos_creat

Creat a file in a DOS environment as opposed to a UNIX type environment. The attribute byte is the same as described in the DOS Technical Reference Manual. Here are the attributes defined in dos.h:

ATTRIBUTE	***VALUE***	
FA_RDONLY	0x01	Read Only
FA_HIDDEN	0x02	Hidden file
FA_SYSTEM	0x04	System file
FA_LABEL	0x08	Volume label
FA_DIREC	0x10	Sub-directory
FA_ARCH	0x20	Archive bit

dos_get_verify
Returns the state of the DOS verify status. This function will return a non-zero value if VERIFY is on and a zero if the VERIFY is off. For additional information, see the VERIFY command in your DOS Technical Reference Manual.

dos_get_cntl_break
Returns the state of the DOS control break status. Returns a non-zero value if BREAK checking is on and a zero value if BREAK checking is off. For additional information, see the BREAK command in your DOS Reference Manual.

dos_getdiskfreespace
Returns a long integer of available disk space. Drive is an integer value where 0 = default, 1 = A:, 2 = B: 3 = C:, etc...

dos_set_verify
Sets automatic read-after-write verification. This function has the same effect as the MS DOS commands VERIFY ON and VERIFY OFF.

dos_set_cntl_break
Turns the control break checking on or off. A non-zero value for the on/off argument turns control break checking on and a zero value turns off control break checking. This function has the same effect as the BREAK command does for the DOS command processor.

Example:

```
#include <dos.h>
#include <stdio.h>
char buffer[512];
main()
{
   char *offset;
   long result;
   int drive = 0;
   result = dos_getdiskfreespace(drive);
   printf("Default Drive = %ld Bytes free\n",result);
   dos_abs_disk_read(0,1,0,buffer);
   offset = &buffer[3];
   printf("Boot Record OEM = %s\n",offset);
}
```

Return Value:

dos_creat
Returns a DOS handle if the file was successfully created otherwise a -1 and errno is set.

dos_get_cntl_break, dos_get_verify
Return a non-zero value if the status is on and a zero if the status is off.

dos_getdiskfreespace
Returns a long integer of available disk space.

dos_set_verify, dos_set_cntl_break
Does not return a value.

dos_abs_disk_read, dos_abs_disk_write
Returns a 0 on success. A non-zero return value indicates an error. The lower byte will contain the DOS error code. The higher byte will contain the specific BIOS error. See your DOS Technical Reference Manual.

ecvt

Usage:

#include <stdlib.h>
*char *ecvt(double val,int ndig,int *pdecpt,int *psign);*

Description:

Converts the double value into a string and returns a pointer to that string. The number of digits created is given by ndig. The digit string is rounded. The position of the decimal point relative to the left of the first digit in the string is stored in *pdecpt. If *pdecpt is negative, the decimal point is positioned that many places to the left of the string. Into *psign is stored 0 if val >= 0, else a non-zero value. The string is written into a statically allocated area, shared with fcvt(), which is reused at each call.

Example:

```
#include <stdio.h>
#include <stdlib.h>
char *buffer;
int dec, sign;
main()
{
   buffer = ecvt(98.69138432, 5, &dec, &sign);
   printf("buffer = \"%s\", decimal = %d, sign = %d\n",
                  buffer,dec,sign);
}
```

Return Value:

Returns a pointer to the string of digits.

See Also:

fcvt, sprintf

execl - execvp

Usage:

```
#include <process.h>
#include <errno.h> /* for error checking only */
int execl(char *name,char *arg0, arg1,...,argn, NULL);
int execv(char *name,char *argv[]);
int execlp(char *name,char *arg0, arg1,...,argn, NULL);
int execvp(char *name,char *argv[]);
```

Description:

exec functions create/execute a new child process. A successful exec call replaces the parent process in memory with the child process. Sufficient memory must be available for the child process. exec overwrites the current program in memory with a new program and begins execution. *name* specifies the program to execute. If the filename doesn't have a path and it is execlp or execvp and it is not in the current directory, then the environment variable PATH is used to determine which directories are searched for the file. Arguments passed to a subprocess are made up of character strings in the exec call. These form the argument list for the child process. Their combined length (except arg0) must not exceed 128 characters. arg0 is ignored, it is included for compatibility reasons. The argv argument used in the execv and execp is an array of character pointers. The last pointer in argv must be NULL to indicate the end of the array.

Return Value:

The exec functions do not return if the call was successful. If a return occurs, then the return value is -1, indicating an error. The external integer errno will be set to a value indicating the error. The values are declared in errno.h and are defined as follows:

E2BIG The argument list is more than 128 bytes.
ENOENT The file name was not found.

See Also: abort, exit, spawn, system

Example:

```
#include <process.h>
#include <errno.h>
#include <stdio.h>
main(int argc,char *argv[])
{
   char *args[4];
   int result;
   args[0] = "arg0";
   args[1] = "arg1";
   args[2] = "arg2";
   args[3] = NULL;
   switch (argv[1][0])
   {
      case '1':
         execl("myprog.exe","arg0","arg1","arg2",NULL);
         break;
      case '2':
         execlp("myprog.exe","arg0","arg1","arg2",NULL);
         break;
      case '3':
         execv("myprog.exe",args);
         break;
      case '4':
         execvp("myprog.exe",args);
         break;
      default:
      printf("Enter a number between 1 and 4 on
                                 commandline\n");
      exit(1);
   }
   printf("Process myprog.exe not exec.\n");
}
```

exit, _exit

Usage:

#include <stdlib.h>
void exit(int exitstatus); **/* ANSI */**
void _exit(int exitstatus);

Description:

exit calls all static destructors (for C++ programs), flushes all output buffers, closes all output files and returns to MSDOS with an exit status given by exitstatus.

_exit does not call the static destructors or flush the buffers, but immediately returns to MSDOS.

exit is the preferred function for C++ programs.

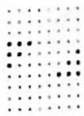
exitstatus is normally a zero to indicate a normal end of program and non-zero to indicate an error. Only the lower order byte of exitstatus is returned to the parent process. The exit status can be referenced by the name ERRORLEVEL in batch files and the spawn return values.

Return Value:

No values are returned.

See Also:

abort



Example:

```
#include <stdlib.h>
#include <stdio.h>
main(int argc,char *argv[])
{
FILE *fp;
if(argc > 1)
   {
      fp = fopen(argv[1],"r");
      if(fp == NULL)
      {
         fprintf(stderr,"Can't open file \"%s\"\n",
                                          argv[1]);
         exit(1);
      }
   }
else
   {
      fprintf(stderr,"No file specified\n");
      exit(1);
   }
}
```

exp

Usage:

#include <math.h>
double exp (double x); **/* ANSI */**

Description:

exp generates the natural exponent of the argument x.

Example:

```
#include <math.h>
main()
{
    double dnum, xpnt;
    dnum = 3.1415926;
    xpnt = exp(dnum);
    printf("The exp(%f) = %f\n",dnum,xpnt);
}
```

Return Value:

Returns the real exponential function e to the x. If overflow, errno is set to ERANGE. If underflow, exp returns 0 and does not set errno.

See Also:

log, pow, sqrt

fabs

Usage:

#include <math.h>
double fabs(double x); **/* ANSI */**

Description:

Returns the absolute value of the floating point argument x.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double numbr, absval;
   numbr = -1.0;
   absval = fabs(numbr);
   printf("The fabs of (%f) is %f\n",numbr,absval);
}
```

Return Value:

Returns the absolute value of its double argument.

See Also:

abs, labs

farcalloc - farrealloc

Usage:

```
#include <dos.h>
void far *farmalloc(unsigned long size);
void far *farcalloc(unsigned long number
                    unsigned long size);
long farcoreleft(void);
void farfree(void far *memblock);
void far *farrealloc(void far *memblock,
                     unsigned long newsize);
```

Description:

farmalloc	Allocates memory size long from the far heap.
farcalloc	Allocates memory for an array containing number, each size bytes long from the far heap.
farcoreleft	Returns the amount of memory in the far heap.
farfree	Releases a memblock of previously allocated far memory.
farrealloc	Adjusts the size of a previously allocated memblock to newsize.

Note that all available RAM can be allocated, memblocks larger than 64k can be allocated, and far pointers are used to access memblocks.

These functions are direct hooks into MS DOS. They are much slower than malloc, etc. They can also prevent the heap from expanding fully in S, M, T models (so expand the heap first). Pointers allocated with farxxx() must be freed before the program terminates otherwise the memory will stay allocated until the machine is rebooted.

Example:

```
#include <stdio.h>
#include <dos.h>
main()
{
char far *memblock;
long allocate = 65000;
   printf("%lu bytes memory free\n",farcoreleft());
   memblock = farmalloc(allocate);
   if(memblock == NULL) {
      printf("failed to allocate memory-block\n");
      exit(1);
   }
   printf("%lu bytes allocated, ",allocate);
   printf("%lu bytes memory free\n",farcoreleft());
   allocate /= 2;
   memblock = farrealloc(memblock, allocate);
   printf("memory now reallocated to %lu bytes, ",
                                          allocate);
   printf("%lu bytes memory free\n",farcoreleft());
   farfree(memblock);
   printf("memory-block now freed, ");
   printf("%lu bytes memory free\n",farcoreleft());
}
```

Return Value:

farmalloc	Returns a far pointer to the allocated memblock, or NULL if not enough space exists.
farcalloc	Returns a far pointer to the allocated memblock, or NULL if not enough space exists.
farcoreleft	Returns the amount of memory in the far heap.
farfree	Returns nothing.
farrealloc	Returns the address of the reallocated memblock. This may be different from the original memblock.

See Also: calloc, free, malloc, realloc

_farptr_norm
_farptr_fromlong
_farptr_tolong

Usage:

```
#include <dos.h>
void far *_farptr_norm(void far *ptr);
void far *_farptr_fromlong(long value);
long _farptr_tolong(void far *ptr);
```

Description:

_farptr_norm normalizes a far pointer. That is, it adjusts the segment and offset so that the offset is < 16. *_farptr_fromlong* converts from a long to a normalized far pointer. The long is taken to be a linear address from 0. *_farptr_tolong* converts from a far pointer to long which is the address that the far pointer represents.

Return Value:

_farptr_norm returns the normalized far pointer. *_farptr_fromlong* returns the normalized far pointer. *_farptr_tolong* returns the long.

Example:

```
#include <stdio.h>
#include <dos.h>
main()
{
   void far *pin, far *pout;
   long value;
   pin = MK_FP(0x1100,0x2678);
   pout = _farptr_norm(pin);
   printf("%lp normalized should be 1367:0008,"
      " it actually is %lp\n", pin,pout);
   value = 0xB7543;
   pout = _farptr_fromlong(value);
   printf("Converting 0x%lX to a far pointer should be"
      " B754:0003,\nit actually is %lp\n", value,pout);
   pin = MK_FP(0xB754,0x0013);
   value = _farptr_tolong(pin);
   printf("Converting %lp to a long should be 0xB7553,"
      "\nit actually is 0x%lX\n",pin,value);
}
```

fclose

Usage:

#include <stdio.h>
*int fclose(FILE *fp);* **/* ANSI */**

Description:

fclose closes the file associated with fp. The output buffer for fp is flushed before closing.

Example:

```
#include <stdio.h>
main()
{
   FILE *fp;
   fp = fopen("file.dta","r");
   if (fp == NULL)
      printf("Datafile not opened\n");
   else
      {
         fclose(fp);
         printf("File file.dta closed using fclose\n");
      }
}
```

Return Value:

fclose returns 0 if the stream successfully closed. A -1 is returned on error.

See Also:

close, fopen, freopen

fcvt

Usage:

#include <process.h>
*char *fcvt(double val,int nfrac,int *pdecpt,int *psign);*

Description:

Same as ecvt except that nfrac specifies the number of digits that appear after the decimal point.

Example:

```
#include <stdio.h>
#include <process.h>
main()
{
   char *buffer;
   int dec, sign;
   buffer = fcvt(98.69138432, 5, &dec, &sign);
   printf("buffer = \"%s\", decimal = %d, sign = %d\n",
            buffer,dec,sign);
}
```

Return Value:

Returns a pointer to a static string of digits which is overwritten by each call to ecvt or fcvt.

See Also:

ecvt, sprintf

feof

Usage:

#include <stdio.h>
*int feof(FILE *fp);* **/* ANSI */**

Description:

feof determines if the stream fp is at the end of file.

Example:

```
#include <stdio.h>
#define BUFSIZE 128
char buffer[BUFSIZE];
main()
{
   FILE *fp;
   fp = fopen ("file.dat", "r");
      {
         while ( !feof(fp) )
            fgets(buffer, BUFSIZE, fp);
      }
   printf("Done reading file, EOF found\n",buffer);
}
```

Return Value:

Non-zero (eof flag set) if current position is end. No read operations are allowed after the flag is set. The flag is cleared if rewind, fseek or file is closed. 0 if the flag is clear.

See Also:

clearerr, ferror

ferror

Usage:

#include <stdio.h>
*int ferror(FILE *fp);* **/* ANSI */**

Description:

ferror checks the error flag on the stream fp. The error flag remains set until a clearerr, rewind or fseek is issued or the stream is closed.

Example:

```
#include <stdio.h>
main()
{
   int c;
   if ( (c = fgetc(stdin)) == EOF)
      if (ferror(stdin))
         {
            fprintf(stderr,"Read error\n");
            clearerr(stdin);
         }
   printf("No error\n");
}
```

Return Value:

Returns non-zero if the error flag is set, otherwise it returns a zero.

See Also:

clearerr, feof

fflush

Usage:

#include <stdio.h>
*int fflush(FILE *fp);* **/* ANSI */**

Description:

Flushes the buffer associated with stream fp. If the file is opened for writing, the buffer is written. If the file is opened for reading, the buffer is cleared.

Example:

```
#include <stdio.h>
FILE *fp;
main()
{
   if(!(fp = fopen("file.dat","w"))) return;
   fflush(fp);  /* flush buffer to disk to protect */
                /* data before performing operations */
                     /* that may cause system crash */
}
```

Return Value:

fflush returns a 0 if the buffer was successfully flushed. EOF is returned if an error occurred.

See Also:

fclose, flushall

fgetc

Usage:

#include <stdio.h>
*int fgetc(FILE *fp);* **/* ANSI */**

Description:

fgetc reads and returns the next character from the stream fp.

Example:

```
#include <stdio.h>
main()
{
   int ch;
   fputs("Enter data terminated by Ctrl-Z: ",stdout);
   while((ch = fgetc(stdin)) != EOF) fputc(ch,stdout);
}
```

Return Value:

Returns the character just read on success, or EOF if end-of-file or a read error is encountered.

See Also:

fputc, putchar, getc, getchar, getche, getch.

fgets

Usage:

#include <stdio.h>
*char *fgets(char *str,int n,FILE *fp);* **/* ANSI */**

Description:

Reads characters from stream fp into the string pointed to by str. Reading stops when a newline is read, end-of-file is encountered, a read error occurs, or n-1 characters were read. Newlines are included in the string. The string read is terminated with a 0.

Example:

```
#include <stdio.h>
main()
{
   char buffer[255];
   int buflen;
   char *result;
   buflen = 255;
   fputs("Enter line of data\n",stdout);
   result = fgets(buffer,buflen,stdin);
   if(!result)
      printf("\nEOF or ERROR\n");
   else fputs(buffer,stdout);
}
```

Return Value:

String str if successful. If no characters have been read into str and a read error or EOF is encountered, NULL is returned and the string pointed to by str is unchanged. If a read error occurs, NULL is returned and str contains garbage.

See Also: fputs, gets, puts

fileno

Usage:

#include <stdio.h>
*int fileno(FILE *fp);*

Description:

fileno returns the file descriptor associated with stream fp. This is the same as the MSDOS file handle.

Example:

```
#include <stdio.h>
main()
{
   int fd;
   fd = fileno(stderr);
   printf("The file handle for stderr is %d\n",fd);
   fd = fileno(stdaux);
      printf("The file handle for stdaux is %d\n",fd);
   fd = fileno(stdprn);
   printf("The file handle for stdprn is %d\n",fd);
}
```

Return Value:

fileno returns the file descriptor. There is no error return.

See Also:

fopen, freopen.

filesize

Usage:

#include <io.h>
*long filesize(char *filename);*

Description:

filesize determines the size of a file in bytes. The filename must be an existing file that is not currently open.

Example:

```
#include <io.h>
main()
{
    long size;
    size = filesize ("file.dat");
    printf("The filesize of file.dat is %ld \n",size);
}
```

Return Value:

The length of the file in bytes is returned upon success, otherwise a -1L is returned and errno is set.

See Also:

stat, fstat.

findfirst, findnext

Usage:

#include <dos.h>
*struct FIND *findfirst(char *pathname,int attribute);*
*struct FIND *findnext(void);*

Description:

The system calls findfirst and findnext are used to find files matching a file description possibly containing wild cards (i.e. '*' or '?'). The wild cards can be in the file name or extension, but not the path. The attribute argument is the file attribute of the files to be found. More than one attribute bit can be passed in the findfirst call. The attribute bits are defined in dos.h and are:

ATTRIBUTE	FILE-TYPE
FA_RDONLY	0x01
FA_HIDDEN	0x02
FA_SYSTEM	0x04
FA_LABEL	0x08
FA_DIREC	0x10
FA_ARCH	0x20

A value of 0 for the attribute will find all normal files. A pointer to a static structure (defined in dos.h) is returned upon success, otherwise a NULL is returned. The pointer to the returned structure is static, so if the information is needed after a call to findnext, then the structure must be copied to another buffer. The system call findnext gets the next match of the file name specified in the preceding findfirst call. Findnext must be called only after a findfirst call.

Example:

```
#include <stdio.h>
#include <dos.h>
main()
{
    struct FIND *p;
    printf("Directory listing of *.*\n");
    p = findfirst("*.*",0);
    while(p)
        {
            printf(" %s\n",p->name);
            p = findnext();
        }
}
```

Return Value:

A pointer to a static struct FIND as defined in dos.h is returned on success and a NULL pointer to indicate the end of filename matches or an error.

floor

Usage:

#include <math.h>
double floor (double x); **/* ANSI */**

Description:

floor returns the largest integer (as a double- precision number) not greater than x.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double d;
   d = floor(4.8);
   printf("The floor of 4.8 is %f\n",d);
   d = floor(-4.2);
   printf("The floor of -4.8 is %f\n",d);
}
```

Return Value:

floor returns the double result.

See Also:

ceil, fmod

flushall

Usage:

#include <stdio.h>
int flushall(void);

Description:

flushall flushes the I/O buffers for all open streams. All streams remain open after call to flushall. Any read operation reads new data into the buffers.

Example:

```
#include <stdio.h>
main()
{
    int totalflushed;
    totalflushed = flushall();
}
```

Return Value:

flushall returns the number of open streams.

See Also:

fflush

fmod

Usage:

#include <math.h>
double fmod (double x,double y); **/* ANSI */**

Description:

fmod is used to obtain the floating point remainder of two numbers. It returns x if x is zero, or the number F with the same sign as x, such that x = L * y + F for some integer L, and |F| < |Y|.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double x, y, d;
   x = 5.0;
   y = 3.0;
   d = fmod(x, y);
   printf("fmod (%.2f, %.2f) is %f",x,y,d);
}
```

Return Value:

fmod returns the floating point remainder.

See Also:

ceil, floor, modf

fopen

Usage:

```
#include <stdio.h>
FILE *fopen(char *name,char *mode);          /* ANSI */
```

Description:

fopen opens a file. name gives the file name to be opened. mode is a character string indicating how the file is to be opened. Possible values for mode are:

"**r**"	for reading
"**w**"	for writing (truncates any existing file with the same name)
"**a**"	for appending (if file exists then open for writing at end of file, else create the file)
"**r+**"	for reading and writing
"**w+**"	for reading and writing (if file exists then truncate it, else create it)
"**a+**"	for reading and writing (if file exists then position at end of file, else create it)

In addition, a 'b' may be appended to the mode string to indicate that the file is to be opened in binary mode (the default is text mode). If a file is opened for reading and writing, only reads or only writes can be done at any one time. To switch from reading to writing, or vice versa, an fseek must be performed on the stream, unless during input an EOF was read.

Example:

```
#include <stdio.h>
main()
{
   FILE *fp;
   fp = fopen("file.dat","r");
   printf("Open text file\n");
   if(fp) fclose(fp);
   fp = fopen("CON","wb");
   printf("Sound bell\n");
   fputc(0x07,fp);
   if(fp) fclose(fp);
   fp = fopen("CON","rb");
   printf("Enter string\n");
   if(fp && fgetc(fp) == 3)
      printf("\n Ctrl-C entered\n");
   if(fp) fclose(fp);
   fp = fopen("PRN","w");
   printf("Write to printer\n");
   if(fp) fclose(fp);
   fp = fopen("file.dat","a+");
   printf("ASCII update file\n");
   if(fp) fclose(fp);
}
```

Return Value:

fopen returns a FILE pointer to an open file. A NULL pointer value indicates an error.

See Also:

fclose, freopen, open

FP_OFF, FP_SEG

Usage:

#include <dos.h>
*unsigned FP_OFF(int far *long_pointer);*
*unsigned FP_SEG(void far *long_pointer);*

Description:

FP_OFF and FP_SEG are used to split far pointers into their offset and segment parts. These functions are implemented as macros.

Example:

```
#include <dos.h>
#include <stdio.h>
main()
{
   char far *p;
   unsigned int segment, offset;
   p = MK_FP(0xb000,0);
   segment = FP_SEG(p);
   offset = FP_OFF(p);
   printf("The mono video segment:offset is at
          %04x:%04x \n", segment,offset);
}
```

Return Value:

FP_SEG It returns the 16-bit segment value of the far pointer.

FP_OFF Returns the 16-bit offset of the far pointer.

See Also

MK_FP

fprintf

Usage:

#include <stdio.h>
*int fprintf(FILE *stream,const char *format,...);*

/* ANSI */

Description:

Sends the formatted output to the specified stream. The format is the same as printf, see printf for description.

Example:

```
#include <stdio.h>
main()
{
   fprintf(stdout,"This is an example \n");
}
```

Return Value:

Returns the number of characters written or a negative value on error.

See Also:

printf, sprintf, vfprintf

fputc

Usage:

#include <stdio.h>
*int fputc(int c,FILE *fp);* **/* ANSI */**

Description:

fputc writes the character c to the stream fp.

Example:

```
#include <stdio.h>
FILE *fp;
main()
{
   char *line = "This is an example ";
   fp = stdout;
   while(*line)
      fputc(*line++, fp);
}
```

Return Value:

fputc returns the last character output to the stream. An EOF is returned on error.

See Also:

fgetc, getc, getchar

fputs

Usage:

#include <stdio.h>
*int fputs(char *s,FILE *fp);* **/* ANSI */**

Description:

fputs writes the string s (excluding the terminating 0) to the stream fp.

Example:

```
#include <stdio.h>
main()
{
   if (fputs ("Hello World\n", stdout) == -1)
   fprintf(stderr, "Output error\n");
}
```

Return Value:

Returns 0 if successful, non-zero if a write error occurred.

See Also:

fgets, gets, puts

fread

Usage:

#include <stdio.h> **/* ANSI */**

*int fread(void *p,unsigned sizelem,unsigned n,FILE *fp);*

Description:

Reads n elements from stream fp into the array that p points to. sizelem is the number of bytes in each element.

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    char *dest;
    int size = 1;
    int number = 256, numread;
    FILE *fp;
    if((fp = fopen("file.dat","r")) == NULL)
       return;
    dest = calloc(256,1);
    numread = fread(dest,size,number,fp);
    printf("Total read %d\n",numread);
    fprintf(stdout,"Data read\n %s",dest);
    free(dest);
}
```

Return Value:

fread returns the number of complete elements actually read. If an error or an end of file is encountered, it will return less than n.

See Also: fwrite, read

free

Usage:

#include <stdlib.h>
*int free(void *p);* **/* ANSI */**

Description:

free releases memory pointed to by p. p must have been allocated using calloc, malloc, or realloc. p can be NULL, in which case free does nothing. Nb. free will return void if the strict ANSI compiler flag is used.

Caution: do not free data items more than once or refer to data after it has been freed.

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   char *p;
   if((p = malloc(1000)) == NULL);
      printf("Unable to allocate memory\n");
         if (free(p)==-1)
            abort(); /* Something is terribly wrong */
}
```

Return Value:

Returns 0 if successful or -1 if the heap is corrupted or p is invalid. A return of -1 is indicative of a serious program bug and the program may shortly crash. free will not always detect attempts to free wild pointers.

See Also: calloc, malloc, realloc

freopen

Usage:

#include <stdio.h>
*FILE *freopen(char *name,char *mode,FILE *fp);*

/* ANSI */

Description:

Closes file indicated by fp. Errors while closing the file are ignored. Open a new file and associate the stream fp with it. Name and mode mean the same as in fopen.

Example:

```
#include <stdio.h>
FILE *fp;
main()
{
   fp = freopen("file.dat", "w+", stdout);
   if(fp == NULL)
      fprintf(stderr,"error on freopen\n");
   else
         fprint(fp,"This data will go to file.dat\n");
}
```

Return Value:

freopen returns fp if successful, otherwise a NULL.

See Also:

fclose, fopen, open

frexp

Usage:

#include <math.h>
*double frexp(double value,int *eptr);* **/* ANSI */**

Description:

Splits the double value into x and e such that (value == x * (2 to the e) and x > .5 and x <= 1.0). e is stored in *eptr and x is returned. If value is 0.0, then both x and e are 0.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double d1, d2;
   int i;
   d2 = 15.3;
   d1 = frexp(d2, &i);
   printf("d1 = %f si = %d\n",d1,i);
}
```

Return Value:

frexp returns the mantissa x.

See Also:

ldexp, modf

fscanf

Usage:

#include <stdio.h>
*int fscanf(FILE *fp,char *format,...);* **/* ANSI */**

Description:

Reads characters from the input stream fp. Characters read are converted according to the format string and the values created are stored through the arg pointers.

Note that args are pointers to where values are to be stored. See scanf.

Example:

```
#include <stdio.h>
char fst[10], *lst = "      ";
int ttl;
main()
{
   int rt;
   rt = fscanf(stdin, "%s %s %d",fst,lst,&ttl);
   printf("Values read %s %s %d\n",fst,lst,ttl);
   return rt = 3;
}
```

Return Value:

Number of successful conversions. Processing ends when the first conflicting character is encountered or when EOF is read from stream. 0 means no fields are converted.

See Also:

scanf

fseek

Usage:

#include <stdio.h>
#include <io.h>
*int fseek(FILE *fp,long offset,int origin);*

/* ANSI */

Description:

Sets the file position associated with the stream fp. Offset is the signed offset in bytes relative to what is specified by origin.

Values for origin are defined in io.h:

SEEK_SET	beginning of file
SEEK_CUR	current position
SEEK_END	end of file

If the file is opened in text mode, offset can only be a value returned by ftell and origin can only be 0, or offset must be 0. If an ungetc was done immediately preceding a fseek, the ungetc is undone. If the file was opened in read/write mode (see fopen), following the fseek either reading or writing may be performed.

Example:

```
#include <stdio.h>
#include <io.h>
main()
{
   FILE *fp;
   fp = fopen("file.dat","r+");
   fseek(fp, 0L,SEEK_END);   /* Move to end of file */
   fseek(fp, 0L,SEEK_SET);
                        /* Move to beginning of file */
   fseek(fp, 200L,SEEK_SET);  /* Move to offset 200 */
}
```

Return Value:

fseek returns 0 if the pointer was successfully moved. fseek returns !=0 if an error occurred.

See Also:

ftell, lseek

fstat

Usage:

#include <sys\stat.h>
*int fstat(int fd,struct stat *buf);*

Description:

fstat function gets information about an open file fd and stores it in the structure that buf points to. The structure stat contains the following fields:

st_dev Drive number of disk containing file fd or, value of fd if fd is a device.

st_mode Bit mask containing mode information on the open file.

S_IFCHR	Set if fd refers to a *device*
S_IFREG	Set if fd refers to an ordinary file
S_IREAD	Set if file is open for reading.
S_IWRITE	Set if file is open for writing.

st_nlink Always 1

st_rdev Same as st_dev

st_size Size of open file in bytes

st_atime Time of last modification

st_mtime Same as st_atime

st_ctime Same as st_atime

There are three other fields st_ino, st_vid, st_gid in the stat structure not mentioned that have no meaningful values under DOS. They are set to 0. They are provided for UNIX compatibility.

Note that fstat cannot be called from a C++ file, see the note under stat.

Example:

```
#include <dos.h>
#include <sys\stat.h>
#include <stdio.h>
#include <io.h>
#include <time.h>
#include <errno.h>
struct stat buf;
int fh, result;
main()
{
   fh = open("file.dat",O_RDONLY);
   result = fstat(fh,&buf);
   if(result != 0)
      printf("Bad file handle\n");
   else
      {
         printf("File size : %ld\n",buf.st_size);
         printf("Drive number : %d\n",buf.st_dev);
         printf("Time modified : %s",
                  ctime(&buf.st_atime));
      }
}
```

Return Value:

fstat returns a value of 0 if file information is successfully obtained. If a value of -1 is returned errno is set to EBADF indicating a bad file handle.

See Also:

stat, findfirst, findnext, isatty

ftell

úíœæ

#include <stdio.h>
*long ftell(FILE *fp);* **/* ANSI */**

Description:

ftell returns the current position in the file associated with the stream fp. If the file is opened in text mode, the returned value may not accurately reflect the number of bytes actually read or written.

Example:

```
#include <stdio.h>
long position;
FILE *fp;
main()
{
   fp = fopen("file.dat","a");
   fprintf(fp,"Sample data string\n");
   position = ftell(fp);        /* get file position */
   printf("Position of file pointer: %ld\n",position);
   fclose(fp);
}
```

Return Value:

ftell returns the current file position. A -1L is returned if an error occurred and errno is set.

See Also:

fseek, isatty

fwrite

Usage:

#include <stdio.h> **/* ANSI */**

*int fwrite(void *buffer,unsigned sizelem, unsigned n,FILE *fp);*

Description:

The function fwrite writes n elements of sizelem bytes each pointed to by buffer to the stream fp.

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    char *buffer;
    unsigned sizelem,n;
    FILE *fp;
    int wrtn;
    buffer = calloc(255,1);
    sizelem = 1;
    n = 255;
    fp = fopen("file.dat","a");
    strcpy(buffer,"\nSample data\n");
    wrtn = fwrite(buffer,sizelem,n,fp);
    free(buffer);
}
```

Return Value:

The function fwrite returns the number of complete elements actually written which may be less than n if an error occurred.

See Also: fread, write

getc, getchar, getche, getch

Usage:

```
#include <stdio.h>          /* defines getc() and getchar() */
#include < conio.h >        /* defines getche() and getch() */
int getc(FILE *fp);                                  /* ANSI */
int getchar(void);                                   /* ANSI */
int getche(void);
int getch(void);
```

Description:

getc obtains one character from the stream fp.

getchar obtains a character from the stream stdin.

getche obtains a character from stdin with echo. This routine will return as soon as a charcter is available and not wait for a <RETURN>. getch is the same as getche except that the character is not echoed to stdout.

getch and getche are direct calls to MSDOS. They bypass the normal buffering done by getchar and getc.

NOTE: Ungetc cannot be used with getch or getche.

Example:

```
#include <conio.h>
#include <stdio.h>
main()
{
   int c;
   printf("Input a character then press return: ");
   c = getchar();
   printf(" Is an example of getchar\n");
   getc(stdin);
   printf("Input a character: ");
   c = getche();
   printf(" Is an example of getche\n");
   printf("Press a function key and then a character");
   c = getch();
   printf("\nCharacter [%c] is an example of getch",c);
}
```

Return Value:

Each of these functions return the next character read. getc and getchar return a value of EOF on error. getch and getche never return an error.

See Also:

fgetc, putc, putchar, ungetc

getcwd

Usage:

#include <direct.h>
*char *getcwd(char *buffer,int length);*

Description:

Returns the drive letter and path name of the current directory and stores it in buffer. If buffer is NULL, getcwd() will malloc() *length* bytes. *length* is the size of the buffer and must be big enough to hold the path name including the terminating 0.

Example:

```
#include <direct.h>
#include <stdio.h>
main()
{
   char buff[66];
   if(getcwd(buff,sizeof(buff)) == NULL)
      perror("getcwd error");
   else
      printf("%s",buff);
}
```

Return Value:

Returns *buffer* on success. A return value of NULL indicates an error and errno is set to:

ENOMEM	(out of memory)
ERANGE	(pathname longer than length-1 characters)

See Also:

chdir, mkdir, rmdir

getDS

Usage:

#include <io.h>
int getDS(void);

Description:

getDS returns the value of the data segment register.

Example:

```
#include <stdio.h>
#include <io.h>
main()
{
   unsigned int data_seg;
   data_seg = getDS();
   printf("The current contents of DS is %x\n",
            data_seg);
}
```

Return Value:

The current value in the data segment register DS is returned.

See Also:

segread

getenv

Usage:

#include <stdlib.h>
*char *getenv(char *name);* **/* ANSI */**

Description:

Getenv searches the environment list for a string of the form <NAME>=<value> and returns a pointer to the value string, if such a string is present. The <NAME> string should be capitalized.

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   char *path;
   path = getenv("PATH");
   printf("PATH = %s\n",path);
}
```

Return Value:

Returns a pointer to the value string of an environment variable. A return value of NULL indicates the name was not found in the environment.

gets

Usage:

#include <stdio.h>
*char *gets(char *str);* **/* ANSI */**

Description:

Read characters from stdin into the string str until a newline is read or an end-of-file is encountered. Newlines are not written to the string. The string is terminated with a 0. str must be large enough to hold the resulting string.

Example:

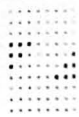
```
#include <stdio.h>
main()
{
    char buffer[125];
    printf("Type in a string: ");
    gets(buffer);
    printf("The string typed was: %s\n",buffer);
}
```

Return Value:

Returns str if successful. A NULL is returned if an end-of-file is encountered and no characters have been written to str, or if a read error occurs.

See Also:

fgets, puts



hypot

Usage:

#include <math.h>
double hypot(double x,double z);

Description:

Calculates the length of the hypotenuse of a right triangle with sides of length x and y. The hypotenuse is the value obtained by taking the square root of the sums of the squares of x and y (sqrt(x*x+y*y)).

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double x,y;
   printf("Enter floating point x: ");
   scanf("lf",&x);
   printf("Enter floating point y: ");
   scanf("lf",&y);
   printf("The hypotenuse of [%f, %f] is
                        [%f]\n",x,y,hypot(x,y));
}
```

Return Value:

The hypotenuse of x and y.

index

Usage:

*char *index(char *s,int c);*

Description:

index searches for the first occurrence of the character c in the string s. This function is identical to strchr.

Example:

```
#include <stdio.h>
#include <string.h>
char string2[50] = " 2 Example string ";
main()
   {
      printf("\nindex example %s]\n",
               index(string2,'s'));
   }
```

Return Value:

Returns a pointer to c if c is found in the string or a NULL if c was not in the string.

See Also:

strchr, strcpy, strcmp, strrchr

inp, inpw

Usage:

#include <dos.h>
int inp(int port_address);
int inpw(port_address);

Description:

These are C interfaces to the 8086 hardware ports using the "in" and "out" 8086 instructions.

inp: Reads a byte from the specified port. The high order byte is zeroed.

inpw: Reads a word from the specified port.

The compiler will generate inline code for these. The real library functions can be called if *#undef inp* and *#undef inpw* are inserted in the source file after *#include <dos.h>*.

Return Value:

The value read from the port is returned from inp and inpw.

See Also:

outp, outpw

Example:

```
    /* This function will turn off the MDA cursor */
#include <dos.h>
int porta,portb,bytea,byteb;
main()
{
   char result;
   porta = 0x3b4;
   portb = 0x3b5;
   bytea = 10;
   byteb = 32;
   result = inpw(porta);
   printf("The value from port #%x is %x hex\n",
            porta,result);
   outp(porta,bytea);
   outp(portb,byteb);
}
```

```
    /* This function will turn on the MDA cursor */
#include <dos.h>
int porta,portb,bytea,byteb;
main()
{
   porta = 0x3b4;
   portb = 0x3b5;
   bytea = 10;
   byteb = 11;
   outp(porta,bytea);
   outp(portb,byteb);
}
```

Interrupt Package

Usage:

```
#include <int.h>
void int_getvector(unsigned vector,unsigned *poffset,
                   unsigned *psegment);
int int_intercept(unsigned vector,
        int(*funcptr) (struct *INT_DATA *pd), unsigned stacksize);
void int_off(void);
void int_on(void);
int int_restore(unsigned vector);
void int_setvector( unsigned vector,unsigned offset,
                   unsigned segment);
```

Description:

The interrupt package is designed to be a general purpose 8086 interrupt handling package. New interrupt handlers can be defined for each vector. The C function handling the interrupt controls whether the interrupt is chained to the previous interrupt vector or not.

int_intercept links a standard C function to an interrupt vector for handling the interrupt when the interrupt occurs. The funcptr is a pointer to the C function that will handle the interrupt. The vector argument defines which 8086 interrupt vector the function will be attached to (0..255). The stacksize argument is the number of bytes of memory allocated for the stack within the interrupt handler. Stacksize should be a minimum of 256 bytes

If a zero value is returned from the interrupt handler (*funcptr)() then the previous interrupt vector for this vector is called else a return from interrupt is performed.

Values in registers can be read/written through the pointer (*pd) to the INT_DATA struct which is passed to (*funcptr)().

There are several rules that must be followed when handling interrupts in C. They are:

1) No MSDOS or BIOS calls may be made during the handling of the interrupt, if the interrupt could occur when MSDOS or BIOS is executing. This is because MSDOS and BIOS are not re-entrant.
2) Do not let the interrupt handler, or any function called by it, use more than the allocated stack space.
3) Do not call any function that is not re-entrant. (Unpredictable results will occur.)
4) Do not perform any I/O from within the interrupt handler.
5) When using variables set by interrupt handlers, declare the variables to be "volatile".
6) Turn off stack overflow checking for the handler.

int_off will turn off the 8086 interrupts via a CLI (clear interrupt flag) instruction.

int_on will turn the 8086 interrupts on via an STI (set interrupt flag) instruction. The function int_restore is the complement to the int_intercept function.

int_restore will unlink the first interrupt handler which was hooked into specified vector via int_intercept. This function must not be called unless an interrupt handler was linked to the vector using int_intercept.

int_getvector obtains the contents of the specified interrupt vector and, splits it into its segment and offset components, and stores each in in the unsigned integers pointed to by *poffset* and *psegment.*

int_setvector installs the address of a user routine supplied in offset and *segment* in the specified interrupt vector

Example:

```
#include <int.h>
#define TRUE 1
volatile int ctrl_c_count = 0;
int do_ctrl_c();
main()
{
   int_intercept (0x23,do_ctrl_c,256);
                                        /* set handler */
   while(count<3)
      printf("Number of Ctrl Cs is %d\n",ctrl_c_count);
   int_restore(0x23);            /* reset old handler */
}
do_ctrl_c()
{
   ++ctrl_c_count;
   return 1;
}
```

Return Value:

The value returned from int_intercept and int_restore indicate success or failure. A zero is returned on success otherwise -1.

WARNING: Intercepting interrupts is an advanced technique and requires a good understanding of MSDOS and the IBM family of PCs to use it successfully.

int86, int86x

Usage:

```
#include <dos.h>                /* register structures */
int int86(int intnum,union REGS *regsin,
          union REGS *regsout);
int int86x(int intnum,union REGS *regsin,
          union REGS *regsout,struct SREGS *segregs);
```

Description:

These functions perform an 8086 software interrupt. Intnum is the interrupt number (0..255).

regsin is a pointer to a structure containing the values of the registers AX,BX,CX,DX,SI and DI to be passed to the interrupt.

segregs is the value of the segment registers to be passed to the interrupt.

regsout is a pointer to a structure into which the return values of the registers will be written.

The REGS and SREGS structures are defined in dos.h. Consult your system hardware and software manuals to determine what the interrupt numbers are and what they do on your machine.

EXAMPLE:

```
/* These functions accept input from the keyboard and
          write the string to the screen via BIOS */
#include <dos.h>
#include <stdio.h>
#define MAXKEY 19
#define BEEP printf("\007")
void print(void);
void biosprt(void);
char buf[MAXKEY],*ptr;
int colm = 30,i;
union REGS r;
main()
{
   system("cls");
   printf("Enter a 20 character string\n\n");
   gets(buf);
      if (strlen(buf)  > 20)
         printf("String to long, truncating\n");
   print();
   BEEP;
}
void print()
{
   r.h.ah = 0x02;
   r.h.bh = 0;
   r.h.dh = 12;
   r.h.dl = colm;
   int86(0x10,&r,&r);
   for(ptr = buf,i = 0; i < = MAXKEY; (biosprt()),i++
                                          ,ptr++)
   ;
}
```

(Continued Overleaf . . .)

(Continued)

```
void biosprt()
{
    r.h.ah = 0x09;
    r.h.al = *ptr;
    r.x.cx = 1;
    r.h.bl = 0x0f;
    r.h.bh = 0;
    int86(0x10,&r,&r);
    r.h.ah = 0x02;
    r.h.bh = 0;
    r.h.dh = 12;
    r.h.dl = ++colm;
    int86(0x10,&r,&r);
}
```

Return Value:

int86 and int86x return the value in the 8086 AX register at the completion of the interrupt. The state of the carry flag can be determined from x.cflag in regsout. Do not use int86 or int86x for interrupts 0x25 or 0x26. Use dos_abs_disk_read/write for those.

See Also:

intdos, bdos

intdos, intdosx

Usage:

```
#include <dos.h> /* register structures */
int intdos(union REGS *regsin,union REGS *regsout);
int intdosx(union REGS *regsin,union REGS regsout,
            struct SREGS *segregs);
```

Description:

These functions perform a DOS system call (int 0x21). Consult your DOS manual for specific function and calling conventions.

regsin is a pointer to a structure containing the values of the registers AX,BX,CX,DX,SI and DI to be passed to the interrupt.

regsout is a pointer to a structure into which the return values of the registers will be written.

The segregs structure contains the segment register values passed to the interrupt for the intdosx. The union REGS and struct SREGS are defined in dos.h.

Example:

```
#include <dos.h>
union REGS inregs, outregs;
struct SREGS segregs;
main()
{
    char far *string = "Print this string$";
    inregs.h.ah = 9;
    inregs.x.dx = FP_OFF(string);
    segregs.ds = FP_SEG(string);
    intdosx(&inregs,&outregs,&segregs);
}
```

Return Value:

The value that was in AX at the end of the interrupt.

See Also:

int86, int86x, bdos

isatty

Usage:

#include <io.h>
int isatty(int fd);

Description:

isatty determines if the handle fd is associated with a terminal, printer, or a serial port.

Example:

```
#include <stdio.h>
#include <io.h>
main()
{
   if (isatty(fileno(stdin)))
      printf("stdin is connected to keyboard\n");
   else
      printf("stdin is connected to a file\n");
}
```

Return Value:

The function isatty returns a non-zero value if the device is a terminal, console, etc. and a zero if not (like a disk file).

See Also:

fstat

is package

Usage:

```
#include <ctype.h>
int isalnum(int c);                    /* ANSI */
int isalpha(int c);                    /* ANSI */
int isascii(int c);
int iscntrl(int c);                    /* ANSI */
int isdigit(int c);                    /* ANSI */
int isgraph(int c);                    /* ANSI */
int islower(int c);                    /* ANSI */
int isprint(int c);                    /* ANSI */
int ispunct(int c);                    /* ANSI */
int isspace(int c);                    /* ANSI */
int isupper(int c);                    /* ANSI */
int isxdigit(int c);                   /* ANSI */
```

Description:

These macros classify characters by table lookup. Each returns a non-zero value for TRUE and a zero for FALSE. isascii is != 0 for all ascii values 0..127. All other macros use only an integer that is between -1 and 255. Values tested outside this range will result in erroneous responses.

These macros are also implemented as functions in the library.

Example:

```
#include <ctype.h>
#include <stdio.h>
main()
{
   printf("Isalnum (%c) = %d\n",'$',isalnum('$'));
   printf("Isalpha (%c) = %d\n','f',isalpha('f'));
   printf("Isascii (%c) = %d\n",0xe4,isascii(0xe4));
   printf("Iscntrl (%c) = %d\n",0x0e,iscntrl(0x09));
   printf("Isdigit (%c) = %d\n",0x32,isdigit(0x32));
   printf("Isgraph (%c) = %d\n",0x0ea,isgraph(0x0ea));
   printf("Islower (%c) = %d\n','A',islower('A'));
   printf("Isprint (%c) = %d\n",0x40,isprint(0x40));
   printf("Ispunct (%c) = %d\n','.',ispunct(','));
   printf("Isspace (%c) = %d\n',' ',isspace(' '));
   printf("Isupper (%c) = %d\n','d',isupper('d'));
   printf("Isxdigit (%c) = %d\n",0x15,isxdigit(0x0d));
}
```

Return Value:

isalnum returns non-zero if c is a letter or a digit.

isalpha returns non-zero if c is a letter.

isascii returns non-zero if c 0..127.

iscntrl returns non-zero if c is a control character (0, 0x1F), or c == 0x7F.

isdigit returns non-zero for the digits 0..9.

isgraph returns non-zero if c is a printing character (excluding the space).

islower returns non-zero if c is a lower-case character.

isprint returns non-zero if c is a printing character (including the space).

ispunct returns non-zero if c is a punctuation character.

isspace returns non-zero for tabs, linefeeds, vertical tabs, form feeds, carriage returns and spaces.

isupper returns non-zero if c is an upper-case character.

isxdigit returns non-zero if c is one of the characters 0..9, A..F or a..f.

See Also:

toascii

itoa

Usage:

#include <stdlib.h>
*char *itoa(int value,char *str,int radix);*

Description:

itoa converts value to a null terminated string using a radix. The radix specifies the base and must be in the range between 2 and 36. If value is negative and the radix is 10, the first character of the stored string is '-'. The result is stored into the string pointed to by str, which must be large enough to hold the result.

Return Value:

itoa returns *str*. There is no error return.

Example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
   char buffer[10];
   int value = 67;
   char *ptr;
   ptr = itoa(value,buffer,2);
   printf("The number %d equals binary =
                          \"%s\"\n",value,buffer);
   ptr = itoa(value,buffer,8);
   printf("The number %d equals octal =
                          \"%s\"\n",value,buffer);
   ptr = itoa(value,buffer,16);
   printf("The number %d equals hex =
                          \"%s\"\n",value,buffer);
}
```

kbhit

Usage:

#include <conio.h>
int kbhit(void);

Description:

The kbhit function checks if a keyboard key has been pressed.

Example:

```
#include <stdio.h>
#include < conio.h >
main()
{
   printf(" Hit any character key when ready\n");
   while(!kbhit())
      ;
   printf("\n The key pressed was (%c)\n",getch());
}
```

Return Value:

Returns non-zero if key pressed, otherwise it returns 0.

labs

Usage:

#include <stdlib.h>
long labs(long i); **/* ANSI */**

Description:

The labs function returns the absolute value of a long integer.

Example:

```
#include <stdlib.h>
#include <stdio.h>
main()
{
   long lng,result;
   lng = -314159L;
   result = labs(lng);
   printf("The absolute value of (%ld) is (%ld)\n"
,lng,result);
}
```

Return Value:

Returns the absolute value of a long integer.

See Also:

abs

ldexp

Usage:

#include <math.h>
double ldexp(double x,int exp); **/* ANSI */**

Description:

The function ldexp calculates the value of x * 2 **exp.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
   double x,y;
   int exp;
   x = 1.5;
   exp = 5;
   y = ldexp(x,exp);
   printf("The ldexp (%f exp %d) = %f\n",x,exp,y);
}
```

Return Value:

ldexp returns x*2 **exp.

See Also:

exp, frexp, log, log10, modf

ldiv

Usage:

#include <stdlib.h> **/* ANSI */**

ldiv_t ldiv(long int numerator, long int denominator)

Description:

The ldiv function divides the numerator by the denominator, returning the quotient and remainder. If the denominator is 0 the program will terminate with an error message. The arguments and the returned values are all type long.

Return Value:

ldiv returns a structure of type ldiv_t, containing both quotient and remainder.

See Also:

div

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   long numer,denom;
   ldiv_t result;
   numer = 3;
   denom = 2;
   printf("The numerator is %ld and the denominator is
                                        %ld\n",
   numer,denom);
   result = ldiv(numer,denom);
   printf("The quotient is %ld \n",result.quot);
   printf("The remainder is %ld \n",result.rem);
}
```

localtime

Usage:

#include <time.h>
*struct tm *localtime(time_t *time);* **/* ANSI */**

Description:

localtime converts a time stored as a time_t value to the structure tm. The time_t value time is the seconds elapsed since 1 January 1970. This value can be obtained from the function time. localtime makes corrections for time zones and possible daylight savings time. The fields of the structure tm are:

```
struct tm {
    tm_sec,                          /* seconds 0..59 */
    tm_min,                          /* minutes 0..59 */
    tm_hour,                     /* hour of day 0..23 */
    tm_mday,                    /* day of month 1..31 */
    tm_mon,                            /* month 0..11 */
    tm_year,                     /* years since 1900 */
    tm_wday,          /* day of week, 0..6 (Sun..Sat) */
    tm_yday,                  /* day of year, 0..365 */
    tm_isdst;         /*  >0 if daylight savings time */
                /* ==0 if not DST, < 0 if don't know */
};
```

Return Value:

localtime returns a pointer to the a static structure tm which is overwritten by each call to localtime().

Example:

```
#include <time.h>
#include <stdio.h>
main()
{
   struct tm *t;
   time_t ltime;
   time(&ltime);
   t = localtime(&ltime);
   printf("The date is %d-%d-%d\n",t->tm_mday,
   t->tm_mon+1,t->tm_year);
}
```

See Also:

time, asctime, ctime, mktime

log, log10

Usage:

#include <math.h>
double log(double x); **/* ANSI */**
double log10(double x); **/* ANSI */**

Description:

log returns the natural logarithm of x. The value of x must be >0.
log10 returns the logarithm base ten of x. (x must be >0).

Example:

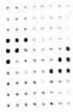
```
#include <math.h>
#include <stdio.h>
main(int argc,char **argv)
{
    double d,result;
    d = atof(argv[1]);
    result = log(d);
    printf("The log of %.2f is %f\n",d,result);
    result = log10(d);
    printf("The log10 of %.2f is %f\n",d,result);
}
```

Return Value:

The log and log10 functions return the logarithm result or domain error if `x<0`, range error if `x==0`.

See Also:

exp, pow, sqrt



longjmp

Usage:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);        /* ANSI */
int setjmp(jmp_buf env);                     /* ANSI */
```

Description:

These functions allow a goto between functions. They are good for dealing with errors or interrupts encountered in low-level subroutines of a program.

setjmp saves the stack environment in the variable env for later use by longjmp.

longjmp restores the environment previously saved by setjmp in jmp_buf and is pointed to by envp. The return value is that of setjmp.

NOTE: The environment must have been previously saved using setjmp by a function that is currently active, and which is the same function or a parent of the function containing the call to longjmp.

After completion of longjmp, program execution continues just as if the corresponding call to setjmp had just returned with value. The value will never be 0. If value is passed as 0, the value 1 will be returned.

Return Value:

setjmp returns a 0. There is no return value for longjmp.

Example:

```
#include < setjmp.h >
#include <stdio.h>
void doscall(void);
jmp_buf environment;
int error_val = -1;
main()
{
   int error_code;
   error_code = setjmp(environment);
   if(error_code != 0)
      {
         printf("Longjmp called\n");
         exit(1);
      }
   printf("Setjmp called\n");
   docall();
}
docall()
{
   longjmp(environment,error_val);
}
```

lseek

Usage:

#include <io.h>
long lseek(int fd,long offset,int mode);

Description:

lseek changes the read/write pointer for a file given by file descriptor fd. Values for mode are:

SEEK_SET	The pointer is moved to offset bytes from the beginning of the file.
SEEK_CUR	Moved to the current location plus offset.
SEEK_END	Moved to the end of the file plus the offset.

Return Value:

lseek returns the offset in bytes of the new position from the beginning of the file. A -1 is returned on error, and errno is set.

See Also:

fseek

Example::

```
#include <stdio.h>
#include <io.h>
#include <dos.h>
main()
{
   int fp;
   long offset,lpos;
   fp = open("file.dat",O_RDWR);
   if(fp <  0) return;
   offset = 0L;
   lpos = lseek(fp,offset,SEEK_SET);
   printf("Current position of seek = %ld\n",lpos);
   offset = 10L;
   lpos = lseek(fp,offset,SEEK_CUR);
   printf("Current position of seek = %ld\n",lpos);
   offset = 50L;
   lpos = lseek(fp,offset,SEEK_END);
   printf("Current position of seek = %ld\n",lpos);
   close(fp);
}
```

malloc

Usage:

#include <stdlib.h>
*void *malloc(unsigned numbytes);* **/* ANSI */**

Description:

Malloc allocates a block of memory that is numbytes in size. If numbytes is 0, NULL is returned.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_INTS 7623
int *memblock;
main()
{
   memblock = malloc(NUM_INTS*sizeof(int));
   if(memblock == NULL)
      printf("Insufficient memory\n");
   else
      printf("Memory allocated\n");
   free(memblock);
}
```

Return Value:

Pointer to the memory block allocated. NULL is returned if not enough memory is available, or if numbytes is 0.

See Also:

calloc, free, realloc, farmalloc

matherr

Usage:

#include <math.h>
*int matherr(struct exception *e);*

Description:

Matherr is called by functions in the math library when errors are detected. Users may use the library supplied matherr function or define their own procedures for handling errors by including a function named matherr in their programs. matherr must be of the form described above. A pointer to the exception structure will be passed to the user supplied matherr function when errors occur. The structure is defined in <math.h> and matherr is available in the library source for an example.

Return Value:

A zero returned from matherr means error was handled correctly, a one returned means the error could not be handled.

Example:

```
#include <stdio.h>
#include <math.h>
#include <string.h>
main()
{
   printf("log(-1) = %e\n",log(-1));
}
int matherr(x)
struct exception *x;
{
   if(x->type == DOMAIN)
      {
         if(strcmp(x->name,"log") == 0)
            {
               x->retval = log(-(x->arg1));
               return(1);
            }
      }
   return(0);
}
```

memchr - memset

Usage: **All Functions /* ANSI */**

```
#include < string.h >
void *memchr(void *buf1,int c,unsigned count);
int memcmp(void *buf1,void *buf2,unsigned count);
void *memcpy(void *buf1,void *buf2,unsigned count);
void *memmove(void *buf1,void *buf2,unsigned count);
void *memset(void *buf, int val,unsigned count);
```

Description:

memchr searches in the buffer, buf, for a byte with the value of c. The search continues for count bytes or until a value of c is encountered.

memcmp compares each successive byte pointed to by pointer buf1 with the corresponding byte pointed to by buf2 until they do not have the same value or until the number of bytes specified in count have been compared. memcmp returns an integer less than, equal to, or greater than zero, depending on whether the last byte compared in the buffer pointed to by buf1 is less than, equal to, or greater than the corresponding byte pointed to by buf2.

memcpy and *memmove* copy the number of characters specified in count from buf2 to buf1. buf1 is returned. Memcpy is faster, but can't handle overlapping moves. Memmove, the slower of the two, can handle overlapping moves.

memset sets the first count characters pointed to by buf to the value specified by val. It returns buf.

Return Value:

memchr returns a pointer to the location of c in buf if found, otherwise a NULL. memcpy, memmove return buf1; memset returns buf. memcmp returns a value:

< 0 buf1 less than buf2.
= 0 buf1 equals buf2.

Example:

```
#include < string.h >
#include <stdio.h>
char buffer[50];
char buffer2[50] = "Sample String\n";
char srccpy[] = "Sample 2 String\n";
char *result;
int reslt;

main()
   {
   printf("memcpy function in process\n");
   memcpy(buffer,buffer2,15);
   printf("memchr function in progress\n");
   result = memchr(buffer,'t',50);
   if(result != NULL)
      printf("Character (t) found at (%d)\n",
               result - buffer +1);
   else
         printf("Error - (t) not found\n");
   printf("memcmp function in progress\n");
   reslt = memcmp(buffer,buffer2,50);
   printf("The result of memcmp is: %s\n",reslt);
   printf("buffer before memmove: %s",buffer);
   result = memmove(buffer,srccpy,sizeof(srccpy));
   printf("buffer after memmove: %s",buffer);
   printf("buffer before memset: %s",buffer);
   result = memset(buffer,'x',6);
   printf("buffer after memset: %s",buffer);
}
```

See Also:

strcmp, strcat, strset, strchr

mkdir

Usage:

#include <direct.h>
*int mkdir(char *pathname);*

Description:

mkdir creates a new directory as specified by the pathname argument. If pathname contains more than one directory component, then only the last component may be new. All preceding components must refer to existing directories.

Example:

```
#include <stdio.h>
#include <direct.h>
main()
{
   int result;
   result = mkdir("\\temp");
   if(result == 0)
      printf("Directory \\temp created\n");
   else
      printf("Could not create directory \\temp\n");
}
```

Return Value:

Returns a 0 if the directory was created successfully. A -1 is returned if an error occurred, and errno is set.

See Also:

rmdir, chdir

MK_FP

Usage:

#include <dos.h>
*void far *MK_FP(unsigned seg, unsigned off);*

Description:

Converts the unsigned segment and offset values to a far pointer.

Example:

```
#include <dos.h>
#include <stdio.h>

main()
{
   char far *p;
   unsigned int segment = 0xb800, offset = 0;

   p = MK_FP(segment,offset);
   printf("The CGA video buffer is at %lp\n",p);
}
```

Return Value:

MK_FP returns a far pointer.

See Also:

FP_SEG, FP_OFF

mktime

Usage:

#include <time.h>
*time_t mktime(struct tm *ntime);* **/* ANSI */**

Description:

Converts from struct tm to time_t.

Example:

```
#include <time.h>
#include <stdio.h>

main()
{
    struct tm ntime;
    time_t set;

    time(&set);
    ntime = *localtime(&set);
    printf("Seconds since 1 Jan 1970 is %ld"
            ,mktime(&ntime));
    printf("The time is %s",asctime(&ntime));
}
```

Return Value:

Time in seconds since 00:00:00 1 Jan 1970 Greenwich Mean Time.

See Also:

asctime, ctime, localtime, time

modf

Usage:

#include <math.h>
*double modf(double x,double *ptr);* **/* ANSI */**

Description:

modf returns the signed fractional value of x. The signed integral part of x is stored at the location pointed to by ptr.

Example:

```
#include <math.h>
#include <stdio.h>
main()
{
   double x,y,n;
   x = -41.56789;
   y = modf(x,&n);
   printf("modf(%f) = fractional: %f and integer: %f"
            ,x,y,n);
}
```

Return Value:

The function modf returns the signed fractional portion of x.

See Also:

frexp, ldexp

Mouse Package

Usage:

```
#include < msmouse.h >
int msm_init(void);
void msm_term(void);
void msm_showcursor(void);
void msm_hidecursor(void);
int msm_getstatus(unsigned *curposx, unsigned *curposy);
void msm_setcurpos(unsigned curposx, unsigned curposy);
int msm_getpress(unsigned *count, unsigned *curposx,
                 unsigned *cpuposy);
int msm_getrelease(unsigned *count, unsigned *curposx,
                 unsigned *curposy);
void msm_setgraphcur(int hotx, int hoty, int *pmasks);
void msm_setareax(unsigned minx, unsigned maxx);
void msm_setareay(unsigned miny, unsigned maxy);
void msm_settextcur(int select,int scanstart, int scanstop);
void msm_readcounters(int *countx, int *county);
void msm_signal(unsigned mask, void(*func)(unsigned mask,
                 unsigned state,unsigned curposx,
                 unsigned curposy), void *stack);
void msm_lightpenon(void);
void msm_lightpenoff(void);
void msm_setratio(unsigned ratiox, unsigned ratioy);
void msm_condoff(unsigned upperx, unsigned uppery,
                 unsigned lowerx,unsigned lowery);
void msm_setthreshhold(unsigned speed);
```

Description:

The mouse package is an interface to the Microsoft mouse. This package is intended to be used very closely with the mouse programming manual. Please refer to this document for specific programming details.

Example:

```
#include <stdio.h>
#include <msmouse.h>
main()
{
   if (msm_init() == -1)
      {
      printf("Mouse initialization succeeded\n");
      msm_showcursor();
      while (1)
         {
         int status;
         unsigned x,y;
         status = msm_getstatus(&x, &y);
         if (status & LEFT_BUTTON)
            {
            msm_hidecursor();
            printf("x = %u, y = %u\n", x,y);
            msm_showcursor();
            }
            if (status & RIGHT_BUTTON)
               break;
         }
      msm_term();
      }
   else
      printf("Mouse initialization failed\n");
}
```

Return Value:

Mouse interface functions are covered in the mouse manual under programming.

open

Usage:

#include <io.h>
*int open(char *pathname,int rwmode);*

Description:

Opens a file given by pathname. Rwmode specifies the mode for which the file is opened. Rwmode values are found in dos.h and are:

O_RDONLY	read
O_WRONLY	write

Return Value:

Returns the file descriptor (fd) if successful, -1 if an error occurred. errno is set upon error.

See Also:

close, creat, dos_creat, fopen

Example:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>
#include <sys\stat.h>
main()
{
    char *fname;
    unsigned int mode;
    int fd;
    mode = O_RDONLY;
    fname = "file.dat";
```

(continued overleaf...)

(continued)

```
    fd = open(fname,mode,0);
    if(fd == -1)
       perror("Error opening file");
    else
       {
       printf("\nfile: %s opened for reading\n",fname);
       close(fd);
       }
    mode = O_WRONLY;
    fname = "CON";
    fd = open(fname,mode,0);
    if(fd < 0)
       printf("\nError opening console for write:%s\n",
                                                 fname);
    else
       {
       printf("\nfile: %s opened for write\n",fname);
       close(fd);
       }
    mode = O_WRONLY | O_CREAT;
    fname = "file.dat";
    fd = open(fname,mode);
    if(fd == -1)
       perror("Error opening file");
    else
       {
       printf("\nfile: %s opened for output\n",fname);
       close(fd);
       }
}
```

outp, outpw

Usage:

#include <dos.h>
void outp(int port_address,int value);
void outpw(int port_address,int value);

Description:

These are C interfaces to the 8086 hardware I/O ports using the "in" and "out" 8086 instructions.

outp	writes the least significant byte of value to the specified port.
outpw	writes the word value to the specified port.

If dos.h is #included, the compiler generates inline code for outp and outpw.

See Also:

inp, inpw

Example:

This function will turn the cursor off (IBM MDA)

```
#include <dos.h>
int porta,portb,bytea,byteb;
main()
{
   char result;
   porta = 0x3b4;
   portb = 0x3b5;
   bytea = 10;
   byteb = 32;
   result = inpw(porta);
   printf("The value from port#%x is%x hex\n",
                                    porta,result);
   outp(porta,bytea);
   outp(portb,byteb);
}
```

This function will turn the cursor on (IBM MDA)

```
#include <dos.h>
int porta,portb,bytea,byteb;
main()
{
   porta = 0x3b4;
   portb = 0x3b5;
   bytea = 10;
   byteb = 11;
   outp(porta,bytea);
   outp(portb,byteb);
}
```

Return Value:

There are no return values for outp and outpw.

peek

Usage:

*void peek(unsigned seg,unsigned offset,void *buf, int numbytes);*

Description:

Moves numbytes from the memory specified by seg:offset to the buffer pointed to by buf.

Return Value:

No value is returned.

See Also:

poke

Example:

```
/* this program will accept as command line arguments
            on or off - to turn numlock on or off. */
#include <stdio.h>
#include < string.h >
main(int argc,char *argv[])
{
   char state1,state2;
   int i;
   for( i = 0;i <  argc; i++)
      {
         strupr(argv[i]);
      }
   if( (strncmp(argv[1],"ON",2) == 0))
      {
         printf("Numlock is on\n");
         peek(0,0x417,&state1,1);
         state1 |= 0x20;
         poke(0,0x417,&state1,1);
      }
   else
      if( (strncmp(argv[1],"OFF",3) == 0 ))
         {
            printf("Numlock is off\n");
            peek(0,0x417,&state1,1);
            state1 &= ~0x20;
            poke(0,0x417,&state1,1);
         }
      else
         {
            printf("\nEnter ON or OFF as command line \
argument\n");
            exit(1);
         }
}
```

perror

Usage:

#include <stdlib.h>
*void perror(char *msg);* **/* ANSI */**

Description:

Displays a message on the standard output describing the last error that occurred in a system call or library function call. The argument msg is printed first, then a colon, followed by the error description.

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   FILE *fp;
   char *string = "Error message";
   if((fp = fopen("fild.dat","r")) == NULL)
      perror(string);
   else
      printf("File open for reading\n");
   fclose(fp);
}
```

Return Value:

There is no return value.

See Also:

strerror

poke

Usage:

*void poke(unsigned seg,unsigned offset,void *buf, int numbytes);*

Description:

Moves numbytes from the buffer pointed to by buf, to the memory specified by seg:offset.

Return Value:

No value is returned.

See Also:

peek

Example:

```
                                    /* see peek comments */
#include <stdio.h>
#include < string.h >
main(int argc,char *argv[])
{
   char state1,state2;
   int i;
   for( i = 0;i <  argc; i++)
      {
         strupr(argv[i]);
      }
   if( (strncmp(argv[1],"ON",2) == 0))
      {
         printf("Numlock is on\n");
         peek(0,0x417,&state1,1);
         state1 |= 0x20;
```

(continued overleaf...)

(continued)

```
        poke(0,0x417,&state1,1);
      }
   else
      if( (strncmp(argv[1],"OFF",3) == 0 ))
         {
            printf("Numlock is off\n");
            peek(0,0x417,&state1,1);
            state1 &= ~0x20;
            poke(0,0x417,&state1,1);
         }
      else
         {
            printf("\nEnter ON or OFF as command line \
argument\n");
            exit(1);
         }
}
```

poly

Usage:

#include <math.h>
double poly(double x,int deg,double coeff[]);

Description:

Poly evaluates a polynomial of the form :
(x * coeff[deg] + coeff[deg-1]) * x +...+ coeff[0].

Example:

```
#include <stdio.h>
#include <math.h>
double coeff[4];
main()
{
   double x = 1.2,y;
   coeff[0] = 0.0;
   coeff[1] = 1.0;
   coeff[2] = 2.0;
   coeff[3] = 3.0;
   y = poly(x,3,coeff);
   printf("The polynomial is %f\n",y);
}
```

Return Value:

The double result is returned.

pow

Usage:

#include <math.h>
double pow(double x,double y); **/* ANSI */**

Description:

The function pow returns the value of x to the y power. Domain error if x==0 and y<=0 or x<=0 and y is not an integer.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double x = 3, y = 2, result;
   result = pow(x,y);
   printf("%.2f to the power of %.2f is: %.2f\n",
                                        x,y,result);
}
```

Return Value:

The double result is returned.

See Also:

exp, log, sqrt

printf, fprintf, sprintf

Usage:

```
#include <stdio.h>
int fprintf(FILE *fp,char *format,...);          /* ANSI */
int printf(char *format,...);                    /* ANSI */
int sprintf(char *buffer,char *format,...);      /* ANSI */
```

Description:

fprintf, printf and sprintf are the formatted print routines.

fprintf writes its characters to the file stream fp.

printf writes its characters to stdout.

sprintf to the memory buffer.

Arguments are interpreted according to the zero terminated format string. The format string is a sequence of characters with embedded conversion commands. Characters that are not part of the conversion command are output. Conversion commands consist of:

'%' {flag} [field_width] ['.' precision] ['l' or 'L'] conversion_char

where:

% Always signifies the beginning of a conversion command.

To print a (%) use (%%).

Flag Characters

-	Means left justify the conversion.
+	Means that signed conversions always start with a (+ or -).
Space	Means that for positive conversions, the conversion will start with a space. The + flag overrides the space flag.

#	For (x) or (X) conversions, if the result is non-zero then a 0x or 0X will be added to the front of it.
	
	For (o) conversions, a leading 0 will be added.
	
	For floating conversions (e, E, f, g, G), a decimal point will always appear. If it is g or G, then trailing 0s will not be truncated.

Field_width

Decimal integer controlling the minimum number of characters printed. If the actual number of characters is less than the field_width, it is padded with blanks. If the field_width digit string begins with a 0, it is padded with a 0.

If the field_width is the character (*), the actual field_width value is taken from the next int arg. If the field_width is negative, it is treated as if the (-) flag was given and the absolute value of the field_width is used.

If there are more characters than allowed for by the field_width, then the width is expanded appropriately.

Precision

. Followed by a digit string specifying the precision of the conversion. If no digits appear after the ., then the precision is taken as 0. For integral conversions, this is the minimum number of digits. For (g) and (G), the precision gives the maximum number of significant digits. For (e), (E), and (f), it is the number of digits appearing after the decimal point. For (s), it is the maximum number of characters in a string. If the precision starts with a 0, then the conversion is 0 padded.

l (Lower case L)

If a o,u,x,X,i,d conversion character is specified, then the argument is taken to be a long, except for a p conversion, when the argument is taken to be a far pointer. For all other conversion characters, it is ignored.

L This flag is ignored.

Conversion_char

One of the characters d, i, o, u, x, X, f, e, E, g, G, c, s, p, n, %. Other characters cause undefined behavior.

d,i,o,u,x,X The argument is an integer and it is converted to a string of digits according to the conversion character. o is unsigned octal, u is unsigned decimal, x and X are unsigned hex, i and d are signed decimal. For x, lower-case hex letters are used. For X, upper-case ones are used. If no precision is specified, it defaults to 1. If there are fewer digits than precision, leading spaces are placed before the digits. If the argument is 0 and the precision is 0, no characters are printed.

c The least significant byte of the integer argument is printed as a character.

e,E The argument is a double. It is printed using scientific notation ([-]d.dddddde+-dd). There is one digit before the decimal point and precision digits after. The precision defaults to 6. If the precision is 0, the decimal point is not written. E is used for the exponent instead of e if the E conversion character was specified. A minimum of two digits will appear in the exponent.

f The argument is a double. It is converted to a decimal string of the form [-]dd.dddd. The number of digits after the decimal point is given by the precision, which defaults to 6. If the precision is 0, no fractional digits or decimal points appear.

g,G The argument is a double. It is printed using f or e (or E if G was specified) format, depending on the value of the argument. e will be used if the exponent is <-3 or > the precision. The precision gives the number of significant digits; it defaults to 6. The decimal point appears if followed by a digit; trailing 0s are truncated.

n The argument is a pointer to an int, into which is written the number of characters printed so far. No characters are generated or converted by this.

p The argument is a pointer which is printed as segment:offset for far pointers or as xxxx for near pointers.

s The argument is a pointer to a string. The characters are printed until a 0 is encountered or the number of characters specified in precision are printed. The terminating 0 is not printed. The precision defaults to 32767.

% The % character is printed.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   char *msg = "Integer formats are:";
   int number = 10;
   double num = 345.2356;
   printf("%s Hex: 0%x, Decimal: %d, Octal: %o\n",
   msg,number,number,number);
   printf("\nFloating point precision for PI: %.2f, %.4f,
                                   %.8f\n", PI,PI,PI);
   printf("\n%.4f In exponential format is: %E\n",
                                          num,num);
}
```

Return Value:

Returns the number of characters written. If an output error occurred, then a negative integer is returned.

See Also:

scanf, vprintf, vsprintf, vfprintf

putc, putchar

Usage:

#include <stdio.h>
*int putc(int c,FILE *fp);* **/* ANSI */**
int putchar(int c); **/* ANSI */**

Description:

putc and putchar write the character c to a stream (putc to the stream fp and putchar to the stream stdout).

Example:

```
#include <stdio.h>
main()
{
   char *string = "This is an example of putc()";
   int i, ch;
   for(i=0; (i<28) && ((ch = putc(string[i],stdout))
                                        != EOF) ; i++);
   putchar('.');
   putchar('\n');
}
```

Return Value:

The character just written is returned. An EOF is returned when an error occurs.

See Also:

fputc, getc, getchar

puts

Usage:

#include <stdio.h>
*int puts(char *s);* **/* ANSI */**

Description:

puts writes the string s to stdout (without the terminating 0), then writes a newline to stdout.

Example:

```
#include <stdio.h>
main()
{
   char *string = "Display this string using puts,";
   char *string2 = "Puts inserts a newline character.";
   puts(string);
   puts(string2);
}
```

Return Value:

Returns a positive value if successful, otherwise EOF.

See Also:

fputs, gets

qsort

Usage:

include <stdio.h> **/* ANSI */**

*void qsort(void *base,unsigned int nel, unsigned size,*
*int compar(void *,void *));*

Description:

qsort is an implementation of the quick-sort algorithm. It sorts a table of data.

base	Points to an element at the base of the table.
nel	The number of elements in the table.
size	The size in bytes of one table element.
compar	The name of the comparison function, which is called with two arguments that point to the elements being compared. compar **must** be written by the programmer and must return an integer that is less than, equal to, or greater than zero according to a comparison of the first argument to the second.

Return Value:

No value is returned.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include < string.h >
#define MAXL 10
unsigned char *line[MAXL];
comp(unsigned char **a,unsigned char **b)
{
   return strcmp(*a,*b);
}
main()
{
   int j, k;
   unsigned char buffer[82];
   printf("Enter 10 lines of data\n");
   for(j = 0;j <  MAXL; ++j)
      {
      printf("Line: %d\n",j+1);
      if(!fgets(buffer,80,stdin))
         break;
      line[j] = malloc(strlen(buffer)+1);
      strcpy(line[j],buffer);
      }
   printf("\n\nSort ten lines from stdin:\n");
   qsort(line,j,sizeof(unsigned char *),comp);
   for(k = 0;k <  j; ++k)
      printf("Line: %d%s\n",k+1,line[k]);
}
```

raise

Usage:

#include < signal.h >
int raise(int sig); **/* ANSI */**

Description:

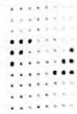
Raise issues a signal to the executing program. The signal type, sig, is discussed in the signal() description. When the signal is called by the raise() function call, the current signal-handling routine will be called. See signal for complete description.

Return Value:

Returns 0 if successful, otherwise returns a non-zero value.

See Also:

signal



Example:

```
#include < signal.h >
#include <stdio.h>
void div_zero(int val)
{
   printf("Divide by zero detected!\n");
   exit(0);
}
main()
{
   float numerator = 3.0;
   float denominator = 0.0;
   if(signal(SIGFPE,*div_zero) == SIG_ERR)
      {
         printf("Could not set SIGFPE!\n");
         abort();
      }
   if(denominator == 0)
      raise(SIGFPE);
   else
      printf("The result of the division is %f\n",
                              numerator/denominator);
}
```

rand

Usage:

#include <stdlib.h>
int rand(void); **/* ANSI */**

Description:

rand returns a random number in the range 0 to 32767. See srand for description.

Return Value:

The integer result.

See Also:

srand

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   int i;
   for(i = 0;i <  20;i++)
   printf("i: %d rand(): %d\n",i,rand());
}
```

read

Usage:

#include <io.h>
*int read(unsigned fd,void *buffer,unsigned length);*

Description:

The system call read gets the next block of characters from the file associated with the file descriptor fd. The number of bytes read into the buffer is specified with the length parameter. The transfer is untranslated (it is a call to MSDOS).

Return Value:

Returns the number of characters actually read, which may be less than length if end-of-file was encountered. If a read error occurs, then a -1 is returned and errno is set.

See Also:

fread, open, write

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <io.h>
#define BYTECOUNT 255
main()
{
   char *buffer;
   int fd,numread,count;
   if((fd = open("file.dat",O_RDONLY)) == -1)
      {
         perror("open failed on file file.dat");
         exit(1);
      }
   buffer = malloc(BYTECOUNT+1);
   for(count = 0;count <  BYTECOUNT;count++)
      buffer[count] = '\0';
   numread = read(fd,buffer,BYTECOUNT);
   printf("\nNumber of characters read %d\n",numread);
   close(fd);
   free(buffer);
}
```

realloc

Usage:

```
#include <stdlib.h>
void *realloc (void *ptr,unsigned size);        /* ANSI */
```

Description:

realloc changes the size of a previously allocated memory block pointed to by ptr. The size of the block after the call to realloc is specified by size. If size is 0, ptr is free'd and NULL is returned. If ptr is NULL, then size is malloc'd and the result is returned. If there is insufficient room to expand the current block a new block will be allocated and the current block released. Existing data will be copied into the new block.

Return Value:

A pointer to the reallocated memory block is returned. If there is insufficient memory for the realloc, NULL is returned (but ptr is not free'd).

See Also:

calloc, free, malloc

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include < string.h >
main()
{
   char *ptr;
   ptr = realloc(NULL,20*sizeof(char));
   strcpy(ptr,"This is part one, ");
   ptr = realloc(ptr,100*sizeof(char));
   strcat(ptr,"This is spart two.");
   printf("\%s\n",ptr);
   realloc(ptr,0)
}
```

rename

Usage:

#include <stdio.h>
*rename (char *oldname,char *newname);* **/* ANSI */**

Description:

Changes the name of a file from oldname to newname. Both oldname and newname may contain drive and path names but both names must refer to the same drive.

Example:

```
#include <stdio.h>
#include <io.h>
main()
{
   int return_code;
   char *oldname,*newname;
   oldname = "data.fil";
   newname = "file.dat";
   return_code = rename(oldname,newname);
   printf("\n%s %s renamed to %s\n",oldname,
   return_code == 0 ? "was" : "was not",newname);
}
```

Return Value:

rename returns a 0 if the file name was successfully changed and non-zero if it was not and errno is set.

rewind

Usage:

#include <stdio.h>
*void rewind(FILE *fp);* **/* ANSI */**

Description:

rewind repositions the file pointer associated with a stream to the beginning of the file. This is equivalent to using `fseek(fp,0L,SEEK_SET)`, with the error flag for fp also cleared.

Example:

```
#include <stdio.h>
main()
{
   FILE *fp;
   char *string = "String one, example string";
   char *string2 = "xxxxxxxxxxxxxx";
   fp = fopen("file.dat","w+");
   fprintf(fp,"%s",&string);
   rewind(fp);
   fscanf(fp,"%s",&string2);
   printf("The value read back is: %s\n",string2);
}
```

Return Value:

rewind returns nothing.

See Also:

fseek

rmdir

Usage:

#include <direct.h>
*int rmdir(char *pathname);*

Description:

rmdir deletes the directory specified by the pathname argument. The directory must be empty and cannot be the root directory or the current working directory. All the intermediate directories must also exist.

Example:

```
#include <stdio.h>
#include <direct.h>
main()
{
   int result;
   result = rmdir("\\temp");
   if(result == 0)
      printf("Directory \\temp removed\n");
   else
      printf("Could not remove directory \\temp\n");
}
```

Return Value:

The function rmdir returns a 0 if the directory was deleted. A return value of -1 indicates an error and errno is set.

See Also:

mkdir, chdir

sbrk

Usage:

*void *sbrk(unsigned int count);*

Description:

sbrk attempts to enlarge the data segment by the number of bytes specified in count. A pointer to the added memory is returned on success, else -1 if error.

For T, S and M models if the variable _okbigbuf was not declared, then all available memory up to 64k is allocated during program start-up and sbrk will always fail (return a -1). If in the program _okbigbuf is declared and initialized to 0, then only the memory required by a program is allocated to the heap and sbrk will be usable.

For C and L models, if sbrk() cannot extend the data segment, it will try to allocate a new one.

sbrk() is an integral part of calloc(), malloc(), and realloc(). Applications should avoid using it.

Return Value:

sbrk returns a -1 if there was not enough memory to satisfy the request and errno is set. Otherwise, a pointer to the memory block is returned.

See Also:

calloc, free, malloc, realloc

Example:

```
#include < string.h >
#include <stdio.h>
void *sbrk(int);
main()
{
   unsigned int count = 100;
   char *ptr;
   ptr = sbrk(count);
   if(ptr == (char *)-1)
      {
         perror("No available space for sbrk\n");
         return;
      }
   strcpy(ptr,"String of data:");
   strcat(ptr," another string added\n");
   fputs(ptr,stdout);
}
```

scanf

Usage:

```
#include <stdio.h>
int scanf(char *format,...);                    /* ANSI */
int fscanf(FILE *fp,char *format,...);          /* ANSI */
int sscanf(char *buffer,char *format,...);      /* ANSI */
```

Description:

These are formatted input routines. The ...s are 0 or more arguments which are pointers to the variables to be set.

sscanf Reads from the input stream fp.
scanf Reads from stdin.
sscanf Reads from the specified buffer. Characters read are converted according to the format string and the values created are stored through arg pointers.

args must be pointers to where values are stored.

The format string consists of:

1. Spaces, tabs and newlines which cause input to be skipped up to the next character which is *not* a whitespace.
2. Other characters, except for % which are matched against the input.
3. A conversion specification, which is as follows:

'%' ['*'] [field-width] [precision] conv_char

This specifies how input characters are to be converted and assigned through the corresponding argument pointers. Conversion continues until a conflicting input character is encountered or the field_width is reached.

* is an assignment suppression flag. It causes the conversion to be performed but the result is ignored. There is no corresponding argument pointer for this.

If there are less argument pointers than conversion specification the results are unpredictable. If there are extra argument pointers the excess ones are ignored.

field_width

is a sequence of decimal digits specifying the maximum number of characters in the field.

precision

L or l (Lower case L)
If 'l' is used with one of the (d, i, o, u, or x)conversion characters, then it indicates that the argument is a pointer to a long rather than a pointer to an int. The l or L flag, when used with the (e) or (f) (f) conversion character, means that the argument is a pointer to a double rather than a pointer to a float.

h Argument is a pointer to a short.

conv_char

The conversion characters are:

d An integer is expected, the argument pointer must be a pointer to an int.

e,f,g E,G A floating point number is expected. The format of the number is the same as in C. The argument pointer must be a pointer to a float (double if 'l' or 'L' is used).

i An integer is expected. If it starts with a 0, it is taken to be octal. If it starts with 0x or 0X, it is hexadecimal. The argument pointer must be a pointer to an int.

o An octal number is expected. The argument pointer must be a pointer to an int.

p A hexadecimal integer is expected. The argument pointer must be a pointer to a pointer.

u An unsigned integer is expected, the argument pointer must be a pointer to an unsigned.

n Through the argument pointer is stored an integer specifying the number of characters read up to this point by this call to scanf().

[A string is expected. Between the [and a closing] will be the characters acceptable to the string. If the [is immediately followed by a ^ , the acceptable characters for the string are all those except the ones between the ^ and the]. The argument pointer must be a pointer to a string. A 0 is appended to the string.

s A string is expected. The argument pointer must be a pointer to a string. The input field extends until a space or newline is read, which is not part of the field. A 0 is appended to the string.

c A character is expected. The argument pointer must be a pointer to a character. If a field_width is specified, then that many characters are read and the argument pointer must point to a character array large enough to hold the result.

% Match the input with a %.

The conversion characters e, g and x may be in capitals, with no difference in meaning. Other conversion characters will cause unexpected results.

Conflicting characters are left unread in the input stream. There is no direct way to determine if suppressed assignments or literal matches succeeded, unless %n is used.

Example:

```
#include <stdio.h>
main()
{
   unsigned int result, val;
   printf("Enter a decimal integer: ");
   result = scanf("%i",&val);
   printf("The decimal number [%d] is [%4x]
                          hexadecimal",val,val);
}
```

Return Value:

The number of assigned input items excluding any assignment supressed conversions is returned. If the end of file is encountered before any assignments are done or before any conflicts occur, an EOF is returned. scanf() normally returns when it reaches the end of the format string.

See Also:

printf, fscanf, sscanf

segread

Usage:

#include <dos.h> / register structures */*
*void segread(struct SREGS *segregs);*

Description:

Reads segment register values and puts them in SREGS.

Example:

```
#include <dos.h>
#include <stdio.h>
struct SREGS segregs;
unsigned code_seg, data_seg, stack_seg, extra_seg;
main()
{
   segread(&segs);
   code_seg = segregs.cs;
   stack_seg = segregs.ss;
   data_seg = segregs.ds;
   extra_seg = segregs.es;
   printf("\nSegment registers currently contain:\n");
   printf("\nCS: %4x\nDS: %4x\nSS: %4x\nES: %4x\n",
      code_seg, data_seg, stack_seg, extra_seg);
}
```

Return Value:

None

See Also:

intdosx, int86x, getDS

setbuf

Usage:

#include <stdio.h>
*void setbuf(FILE *stream,char *buffer);* **/* ANSI */**

Description:

The setbuf function sets the buffering system for bytes read or written to a stream. If the buffer argument is NULL, the stream is unbuffered. If buffer is not NULL, it is taken to be a pointer to the buffer which is to be used for subsequent read and write calls. buffer must point to a character array of size BUFSIZ (defined in stdio.h). The user specified buffer is used then instead of the default system-allocated I/O buffer.

Example:

```
#include <stdio.h>
char buffer[BUFSIZ];
FILE *fp;
main()
{
   fp = fopen("file.dat","r");
   setbuf(fp,buffer);
   printf("Stream has been set to buffer\n");
}
```

Return Value:

None.

See Also:

setvbuf

setjmp

Usage:

#include < setjmp.h >
int setjmp(jmp_buf env); **/* ANSI */**

Description:

See longjmp.

setvbuf

Usage:

#include <stdio.h> **/* ANSI */**

*int setvbuf(FILE *fp,char *buf, int mode,size_t size);*

Description:

setvbuf specifies the type and size of a buffer to be used for a stream. In addition to the function parameters the following global variable affects the behavior of this function:

_okbigbuf

This is used only in the T, S and M memory models, and controls how buffers are allocated when buf is NULL. It is statically initialized to 0 or 1 by the programmer (the default definition in the library sets it to 1). If _okbigbuf is 1 and the memory model is T, S or M: setvbuf() tries to allocate a buffer outside of the data segment. If that fails, and size <= BUFSIZ, then setvbuf() tries to allocate a buffer within the data segment.

A buffer that is outside the data segment is marked by setting _IOBIGBUF in fp->_flags. If _okbigbuf is 0 or the memory model is C or L: setvbuf() tries to allocate a buffer within the data segment. A buffer allocated by setvbuf() is flagged by _IOMYBUF being set in fp->_flags. See the section entitled *Global Variables* for more information on _okbigbuf.

The function parameters are:

fp Stream pointer that is already opened, but before any reads or writes have been done to the stream.

buf Pointer to buffer, or NULL. If NULL, then setvbuf() uses malloc() or paralloc() to try to allocate a buffer of size bytes. If buf is not NULL, it points to a buffer that setvbuf() will cause to be associated with the stream fp.

mode One of:

_IONBF No buffering. The buf and size parameters are ignored. Unbuffered I/O means that: data written is immediately passed on to DOS. When data is read, exactly enough is read.

_IOLBF Do line buffering. The actual I/O is performed when a newline is read or written.

_IOFBF Full buffering. Data is read a full buffer at a time. Data is written only when the buffer is full.

size If buf is NULL, then size is the number of bytes to allocate for the buffer. If buf is not NULL, then size must be the number of bytes in the buffer that buf points to.

Return Value:

If success, the various fields that fp points to are updated to show the buffer and 0 is returned. If there is insufficient memory for the buffer or the mode parameter is invalid, a non-zero result is returned.

Example:

```
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
   FILE *fp;
   static char buf[100]; /* Make stdprn unbuffered */
   setvbuf(stdprn,NULL,_IONBF,0);
   fprintf(stdprn,"unbuffered\n");
   if (argc == 2)                    /* if an argument */
      fp = fopen(argv[1],"w");
   else
      fp = stdout;              /* use standard output */
   if (setvbuf(fp,buf,_IOLBF,sizeof(buf)))
      printf("setvbuf failed\n");
   else
   {
      fprintf(fp,"This is going to fp\n");
      fclose(fp);
   }
}
```

See Also:

setbuf

signal

Usage:

#include < signal.h >
*void (*signal(int sig, void (*func)(int)))(int);*

/* ANSI */

Description:

signal allows a program to define the how signals from the operating system are to be handled. The sig argument must be one of the constants below:

SIGABRT	Abnormal termination
SIGFPE	Floating point error
SIGILL	Illegal instruction
SIGINT	Interrupt CTRL+C
SIGSEGV	Segment violation
SIGTERM	Terminate (control-break)

The macros below are special values for func:

SIG_DFL	Handled in the default manner
SIG_IGN	Ignored the signal.

signal sets the response.

When a signal happens, first the behavior for the signal is reset to SIG_DFL, then the function for that signal is called and sig is passed to it.

Example:

```
#include < signal.h >
#include <stdio.h>
                        /* our signal handler function */
void ctrl_break(int val)
{
   signal(SIGINT,SIG_IGN);
   printf("Press any key to terminate: ");
   getch();
   exit(0);
}
main()
{
   if(signal(SIGINT,(*ctrl_break)) == SIG_ERR)
      {
         perror("Could not set SIGINT!");
         abort();
      }
raise(SIGINT);
}
```

Return Value:

signal returns the previous value of func. A return value of SIG_ERR indicates an error and errno is set to a positive value.

See Also:

raise

sin, sinh

Usage:

#include <math.h>
double sinh(double x); **/* ANSI */**
double sin(double x); **/* ANSI */**

Description:

sin and sinh return the sine and hyperbolic sine of x. For sin, x is in radians.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
    double x = 4.3,d;
    d = sin(x);
    printf("The sine of (%d) is (%d)\n",x,d);
    d = sinh(x);
    printf("The hyperbolic sine of (%d) is (%d)\n",x,d);
}
```

Return Value:

sin returns the sine of the argument x. sinh returns the hyperbolic sine of the argument x.

Sound Package

Usage:

#include < sound.h >
void sound_tone(int cycles,int uptime,int dntime);
void sound_beep(int freq);
void sound_click();

Description:

sound_tone can play tones. The number of cycles and the cycle times are specified. sound_beep causes the speaker to sound a beep. A frequency of 1000Hz can be approximated using 1331 for freq. sound_click causes the speaker to sound a click.

Return Values:

There are no return values.

Example:

```
#include <stdio.h>
#include < sound.h >
main()
{
   int cycles=5000, uptime=50, dntime=50, freq=1331;
   char ch;
   printf("Sound using sound_click\n");
   sound_click());
   printf("Press any key: ");
   ch = getch();
   printf("\nSound using sound_tone\n");
   sound_tone(cycles,uptime,dntime));
   printf("Press any key: ");
   ch = getch();
   printf("\nSound using sound_beep\n");
   sound_beep(freq));
}
```

spawn

Usage:

```
#include <process.h>
int spawnl(int mode,char *pathname,
           char *arg0, *arg1,...,*argn, NULL);
int spawnlp(int mode,char *filename,
           char *arg0, *arg1,...,*argn, NULL);
int spawnv(int mode,char *pathname,char *argv[]);
int spawnvp(int mode,char *filename,char *argv[]);
```

Description:

The spawn system calls create and execute a new child process. More simply, a new program is run then the old one continues. It works similarly to calling a subroutine.

Creating a new subprocess requires memory to be available for the child process to execute in addition to that used by the current program. See the introduction to this section of the manual for a description of the memory usage during a spawn call. The mode flag is ignored. It may be used for future versions. Use a value of 0 for mode.

The filename argument specifies the program to execute. For spawnlp and spawnvp only, if the filename doesn't have a path and it is not in the current directory then the environment variable PATH is used to determine which directories are searched for the file. The argv[0] is just a place holder and is not passed to the subprocess. This is used for compatibility only, but should always be included for portability. It should be the name of the program that is to run.

The command line passed to the spawned program is made up of the character strings in the spawn call, the first being arg1, etc. The combined length of these strings must not exceed 128 characters.

The argv argument used in the spawnv and spawnp is an array of character pointers. The last pointer in argv must be NULL to indicate the end of the list.

Example:

```
#include <stdio.h>
main()
{
   char *args[4];
   args[0] = "ztcl.exe";
   args[1] = "stuff";
   args[2] = "morestuff";
   args[3] = NULL;
   if(spawnv(0,"C:ZTCl.EXE",args) == -1)
      fprintf(stderr,"exec failed!\n");
}
```

Return Value:

The return value indicates the exit status spawned program. A value of 0 indicates that the spawned program executed successfully. A positive value indicates that the spawned program executed, but was aborted or ended in error, the value returned is the exit status of the child process. A negative value indicates that the spawned program did not execute, and errno is set.

See Also:

abort, exit, exec, system

sprintf

Usage:

```
#include <stdio.h>                          /* ANSI */
int sprintf(char *buffer,char *format,...);
```

Description:

Refer to printf

sqrt

Usage:

#include <math.h>
double sqrt(double x); **/* ANSI */**

Description:

sqrt returns the square root of x. If x is negative then a domain error occurs.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double x;
   for(x = 1.0; x < = 20.0; ++x)
      printf("The square root of %.4f is %.4f\n",
                                          x,sqrt(x));
}
```

Return Value:

sqrt returns the double result.

See Also:

exp, log, pow

srand

Usage:

#include <stdlib.h>
void srand(unsigned seed); **/* ANSI */**

Description:

srand seeds the random number generator (rand()) with seed. If srand() is never called, then the default is as if srand(1) was called.

Example:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
   int i;
   srand(9);
   for(i = 0;i < 20;i++)
      printf("i: %d rand(): %d\n",i,rand());
}
```

Return Value:

There is no return value.

See Also:

rand

sscanf

Usage:

#include <stdio.h>
*int sscanf(char *buffer,char *format,...);* **/* ANSI */**

Description:

Refer to scanf

Example:

```
#include <stdio.h>
char *string = "1.24...";
char str[8];
float fp;
int i;
main()
{
   sscanf(string,"%s",str);
   sscanf(string,"%f",&fp);
   sscanf(string,"%d",&i);
   printf("String : %s\n",str);
   printf("Float : %f\n",fp);
   printf("Integer: %d\n",i);
}
```

stat

Usage:

#include <sys\stat.h>
*int stat(char *path,struct stat *buf);*

Description:

stat gets information about a file or directory specified by path and stores it in the structure that buf points to. See fstat for description of struct type stat.

Return Value:

stat returns 0 if the status information is retrieved. On error the function returns -1 and errno is set to ENOENT indicating a bad path.

See Also:

fstat, filesize

Note: stat and fstat cannot be called from a C++ file, because of the name conflict between struct stat and int stat(). If these functions are required, the use of stat.h, fstat() and stat() must be isolated to a C file.

Example:

```
#include <stdio.h>
#include <time.h>
#include <sys\stat.h>
#include <string.h>
#include <stdlib.h>
main()
{
   char *date;
   int ret;
   struct stat buf;
   if((ret = stat("file.dat",&buf))!=0)
      {
         fprintf(stderr,"stat failure error %d",ret);
         abort();
      }
   date = asctime(localtime(&buf.st_ctime));
   printf("\n %s\n",date);
   printf("\n %d mode\n",buf.st_mode);
   printf("\n %ld size\n",buf.st_size);
}
```

strcat - strdup

Usage:

```
#include <string.h>
char *strcat(char *string1,const char *string2);
                                                    /* ANSI */
char *strchr(const char *string,int ch);            /* ANSI */
int strcmp(const char *string1,const char *string2);
                                                    /* ANSI */
int strcmpl(const char *string1,const char *string2);
char *strcpy(char *string1,const char *string2);
                                                    /* ANSI */
size_t strcspn(const char *string1,const char string2);
                                                    /* ANSI */
char *strdup(const char *string);
```

Description:

strcat appends a copy of string2 onto the end of string1. The application code is responsible for ensuring that there is enough space in string to hold the result. It returns string1.

strchr finds the first occurrence of the character ch in string. It returns a pointer to character ch. A NULL pointer is returned if not found. strchr is identical to index().

strcmp compares two strings, character by character. strcmp returns:

< 0	if string1 is less then string2
= 0	if string1 is equal to string2
0	if string1 is greater than string2

strcmpl compares two strings treating upper and lower case characters as the same.

strcpy copies string2 into the object pointed to by string1 including the terminating null '\0'. It returns string1.

strcspn searches string1 for the first occurrence of a character in string2. It returns the length of the initial segment of string1 that consists of characters in string2 not found. If no character appears in string2, then the total length of string1 not counting the null '\0' terminator is returned.

strdup allocates memory with a call to malloc, copies a string into it and returns a pointer to the malloc'd string. It returns NULL if memory cannot be allocated.

Example:

```
#include <stdio.h>
#include <string.h>
char string1[50] = " 1 Example string ";
char string2[50] = " 2 Example string ";
char buffer[50] = " Rubbish - string ";
main()
{
   printf("strcat example [%s]\n",
                        strcat(string1,string2));
   printf("\nstrchr example [%s]\n",
                        strchr(string1,'2'));
   printf("\nstrcmp example [%d]\n",
                        strcmp(string1,string2));
   printf("\nstrcmpl example [%d]\n",
                        strcmpl(string1,string2));
   printf("\nstrcpy example [%s]\n",
                        strcpy(buffer,string1));
   printf("\nstrcspn example [%d]\n",
                        strcspn(string1,"s"));
   printf("\nstrdup example [%s]\n",strdup(string2));
}
```

strerror

Usage:

#include <string.h>
*char *strerror(int errornum);* **/* ANSI */**

Description:

Maps errornum to an error message string and returns a pointer to that string.

Example:

```
#include <stdio.h>
#include <errno.h>
#include <io.h>
#include <dos.h>
#include <string.h>
int fp;
main()
{
   errno = 0;
   if((fp = open("file",O_RDONLY)) == -1)
      {
         printf(strerror(errno));
      }
   close(fp);
}
```

Return Value:

strerror returns a pointer to the error message string.

See Also:

perror

strlen

Usage:

#include <string.h>
*size_t strlen(char *string);* **/* ANSI */**

Description:

Returns the length of the string excluding the terminating 0.

Example:

```
#include <stdio.h>
#include <string.h>
char *string = "Example string";
size_t result;
main()
{
   result = strlen(string);
   printf("The length of string: [%s] is [%d]\n",
                                    string,result);
}
```

Return Value:

Returns the length of the string.

strlwr

Usage:

#include <string.h>
*char *strlwr(char *string);*

Description:

Converts any uppercase characters in the string to lower case.

Example:

```
#include <stdio.h>
#include <string.h>
char *string = "Mixed Case String";
main()
{
   printf("strlwr Example: [%s]\n",strlwr(string));
}
```

Return Value:

Returns string.

See Also:

strupr

strncat - strpbrk

Usage:

```
#include <string.h>
char *strncat(char *string1,const char *string2, size_t n);
                                                    /* ANSI */
int strncmp(const char *string1,const char *string2, size_t n);
                                                    /* ANSI */
char *strncpy(char *string1,const char *string2, size_t n);
                                                    /* ANSI */
char *strnset(char *string,int ch,size_t n);
char *strpbrk(const char *string1,const char *string2);
                                                    /* ANSI */
```

Description:

strncat appends the lesser of n or strlen(string2) characters of string2 onto the end of string1 and adds a terminating null '\0'. It is the user's responsibility to ensure there is enough space in string to hold the result. It string1.

strncmp compares n characters of string2 to string1. The comparison stops after n characters or end of string. strncmp returns:

< 0	if string1 is less then string2
= 0	if string1 is equal to string2
0	if string1 is greater than string2

strncpy Copies the first n characters of string2 into string1. If string2 is longer than string1, the result will not be null '\0' terminated. If string2 is less than n characters, string1 will be padded to n with null characters. It returns string1.

strnset Sets at most n characters of string, to ch. It returns string.

strpbrk Returns a pointer to the first occurrence in string1 of any character from string2, or NULL if no character from string2 exists in string1.

Example:

```
#include <stdio.h>
#include <string.h>
char string[50] = " Example string";
char *string2 = " Add this string";
char buffer[50];
main()
{
   printf("strncat Example   [%s]\n",
                        strncat(string,string2,50));
   printf("\nstrncmp Example [%d]\n", strncmp (string,
                        string2,10));
   printf("\nstrncpy Example[%s]\n",
                        strncpy(buffer,string2,10));
   printf("\nstrnset Example    [%s]\n",
                        strnset(string2,'x',9));
   printf("\nstrpbrk Example [%s]\n",
                        strpbrk(string2,"s"));
}
```

strrchr - strset

Usage:

#include <string.h>
*char *strrchr(const char *string,int ch);* **/* ANSI */**
*char *strrev(char *string);*
*char *strset(char *string,int ch);*

Description:

strrchr() Finds the last occurrence of character ch in string and returns a pointer to it.

strrchr() Returns a NULL pointer if the character is not found.

strrev() Reverses the order of characters in string leaving a terminating null '\0' at the end. Returns string.

strset() Sets all the characters in string to ch except the terminating null '\0'. Returns string.

Example:

```
#include <stdio.h>
#include <string.h>
char string[50] = "Example string";
char *string2 = " Add this string";
main()
{
   printf("strrchr Example [%s]\n",
                        strrchr(string,'i'));
   printf("\nstrrev Example [%s]\n",strrev(string));
   printf("\nstrset Example [%s]\n",
                        strset(string2,' '));
}
```

strspn

Usage:

#include <string.h>
*size_t strspn(const char *string1,const char *string2);*
/* ANSI */

Description:

Returns the length of the initial segment of string1 which consists entirely of characters found in string2.

Example:

```
#include <string.h>
#include <stdio.h>
char *string = "Example:";
char *cmp = "aEx";
main()
{
   int result;
   printf("Length of string in [%s] containing [%s] \
characters\n", string,cmp);
   printf("strspn is [%d]\n",strspn(string,cmp));
}
```

strstr

Usage:

#include <string.h>
*char *strstr(const char *string1,const char *string2);*
/* ANSI */

Description:

Returns a pointer to the first occurrence of string2 within string1.

Example:

```
#include <string.h>
#include <stdio.h>
char *string = " Example: string";
char *cmp = "str";
main()
{
   printf("strstr Example [%s]\n",strstr(string,cmp));
}
```

Return Value:

strstr returns NULL if no occurrence was found. If string2 is of 0 length, returns string1.

strtod - strtoul

Usage:

```
#include <stdlib.h>
double strtod(char *nptr,char **endptr);
                                        /* ANSI */
long strtol(char *nptr,char **endptr,int base);
                                        /* ANSI */
unsigned long int strtoul(const char *nptr,  char **endptr,int base);
                                        /* ANSI */
```

Description:

strtod Converts the ASCII string pointed to by nptr to a double precision value. It will recognize white space then an optionally signed string of digits containing an optional decimal point. It will stop reading the string at the first character that doesn't represent a part of the number. A pointer to that character will be stored in the object endptr if endptr is not NULL.

strtol Converts the ASCII string pointed to by nptr to a long integer-value. It will recognize white space then an optionally signed string of digits. It will stop reading the string at the first character that doesn't represent a part of the number. A pointer to that character will be stored in the object endptr if endptr is not NULL.

If base is zero the first character after the optional sign will determine the base of the conversion. If the first character is 0 and the second character is not 'x' or 'X', then the string is interpreted as octal. If the first character is 0 and the second character is 'x' or 'X' it will be interpreted as a hexadecimal integer.

If the first character is '1' - '9' the string is interpreted as decimal integer. The upper or lower case letters A - Z are assigned the values from 10 through 35. Only letters with values less than the base are permitted. If the base is not 0, the base must be between 2 and 36.

strtoul Converts the ASCII string pointed to by nptr to a unsigned long decimal.
Description - same as strtol().

Return Value:

strtod If correct values would cause an overflow, plus or minus HUGE_VAL is returned according to the sign and errno is set to ERANGE. If an unrecognized character is encountered before a legal character, zero is returned. If underflow 0 is returned and errno is set to ERANGE.

strtol If correct values would cause an overflow then LONG_MAX or LONG_MIN is returned according to the sign and errno is set to ERANGE. If an unrecognized character is encountered before a legal character, zero is returned.

strtoul If correct values would cause overflow ULONG_MAX is returned according to the sign and errno is set to ERANGE. If an unrecognized character is encountered before a legal character, zero is returned.

See Also:

atof, atoi, atol

Example:

```
#include <stdio.h>
#include <stdlib.h>
char *string,*string2;
main()
{
unsigned long result;
long result1;
double result2;
int base;
   string = "3.1415926Stop here";
   result2 = strtod(string,&string2);
   printf("String [%s] strtod = [%f]\n",string,result2);
   base = 8;
   string = "1011013";
   result1 = strtol(string,&string2,base);
   printf("String [%s] strtol [%ld] (base 8)\n",
                                    string,result1);
   base = 2;
   string = "1011013";
   result = strtoul(string,&string2,base);
   printf("String [%s] strtoul [%ld] (base 2)\n",
                                    string,result);
}
```

strupr

Usage:

#include <string.h>
*char *strupr(char *string);*

Description:

Converts any lower case characters in string to upper case.

Example:

```
#include <stdio.h>
#include <string.h>
char *string = "Mixed Case String";
main()
{
   printf("strupr Example [%s]\n",strupr(string));
}
```

Return Value:

Returns string.

See Also:

strlwr

swab

Usage:

#include <string.h>
*void swab(char *source,char *destination,int n);*

Description:

Copies n bytes from source swapping each pair of adjacent bytes. The converted string is stored at destination. The integer n should be even as pairs of characters are swapped.

Example:

```
#include <stdio.h>
#include <string.h>
char *string1 = "badcfehgjilknm";
char *string2 = "..............";
main()
{
   swab(string1,string2,14);
   printf("string1 [%s] string2 [%s]\n",
                                   string1,string2);
}
```

Return Value:

Returns nothing.

system

Usage:

#include <stdlib.h>
*int system(char *string);* **/* ANSI */**

Description:

system causes the string to be given to the DOS command processor as if the string had been typed in at the console. The current program waits until the command is executed then resumes.

There is no way to determine the exit status of programs run using *system*. Use one of the spawn functions if an exit value is required.

Return Value:

If there was not enough available memory to load and run COMMAND.COM or COMMAND.COM is not found, then a -1 is returned, else a 0 is returned.

See Also:

exec, spawn

Note that since a seperate copy of COMMAND.COM is loaded when using system **it cannot be used to set environment variables in the current environment (i.e. that in existance when the program was run) with the SET command. For example, the code:**
system("SET ABC=DEF");
will have no useful effect.

Example:

```
#include <stdlib.h>
main()
{
   system("set  >>path.log");
   system("cls");
   system("type path.log");
   system("pause");
   system("erase path.log");
   system("cls");
   system("ver");
   system("pause");
   system("cls");
   system("dir /w");
}
```

tan, tanh

Usage:

#include <math.h>
double tan(double x); **/* ANSI */**
double tanh(double x); **/* ANSI */**

Description:

These functions return the tangent (tan) and hyperbolic tangent (tanh) of x. x is measured in radians.

Example:

```
#include <stdio.h>
#include <math.h>
main()
{
   double d = .45987;
   printf("The tangent of .45987 is [%f]\n",tan(d));
   d = .495287;
   printf("The hyperbolic tan of .495287 is [%f]\n",
                                          tanh(d));
}
```

Return Value:

Returns the tangent or hyperbolic tangent of x.

time

Usage:

#include <time.h>
*time_t time(time_t *timeptr);* **/* ANSI */**

Description:

The time function returns the current time in seconds elapsed since 00:00:00 1 JANUARY 1970 and stores that value in *timeptr if timeptr is not NULL.

Example:

```
#include <time.h>
#include <stdio.h>
main()
{
   time_t ntime;
   time(&ntime);
   printf("The time is: %s\n",ctime(&ntime));
}
```

Return Value:

time returns the number of seconds since 00:00:00 1 January 1970.

See Also:

ctime, asctime, localtime, mktime

toascii, tolower, toupper

Usage:

```
#include <ctype.h>
int toascii(int c);
int tolower(int c);                    /* ANSI */
int toupper(int c);                    /* ANSI */
```

Description:

toascii Takes any integer value and discards all but the low order seven bits making up an ascii character. If c is already a valid character, then it is returned unchanged.

tolower Converts any integer value c in the range of A - Z to lower case.

toupper Converts any integer value c in the range of a - z to upper case.

These functions are implemented as macros defined in ctype.h and are also included as functions within the library.

Undefining the macros, or not including ctype.h, will cause the library functions to be used.

Example:

```
#include <ctype.h>
#include <stdio.h>
main()
{
   printf("Uppercase character [%c]\n",toupper(0x6d));
   printf("Lowercase character [%c]\n",tolower(0x4d));
   printf("Make ascii character [%c]\n",toascii(0x0e));
}
```

Return Value:

toascii Returns the low 7 bits of c.

tolower Returns c, converted to lower case if it was upper case. Otherwise c is returned unchanged.

toupper Returns c, converted to upper case if it was lower case. Otherwise c is returned unchanged.

ungetc

Usage:

#include <stdio.h>
*int ungetc(int c,FILE *fp);* **/* ANSI */**

Description:

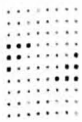
Puts character c back into the input stream fp, where it is read by the next input operation on the stream. If an fseek() is done between an ungetc() and the next read, then the ungotten character is lost. Only one character may be put back between reads. EOF cannot be ungotten.

Return Value:

ungetc returns c if successful and returns an EOF if the character cannot be pushed back.

See Also:

getc, getchar, putc, putchar



Example:

```
#include <stdio.h>
#include <ctype.h>
main()
{
   char ch;
   FILE *stream;
   stream = fopen("file.dat","r");
   while((ch = fgetc(stream)) != EOF)
      if(isspace(ch))
         break;
   ungetc(ch,stream);
   ch = fgetc(stream);
   fclose(stream);
}
```

unlink

Usage:

#include <io.h>
*int unlink(char *filename);*

Description:

unlink deletes the file specified by the string filename.

Example:

```
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
main()
{
   int value;
   char buffer[13];
   char *result;

   printf("Input file to remove: ");
   result = gets(buffer);
   value = unlink(result);
   if(value == 0)
      printf("Erased [%s] from disk\n",result);
   else
      printf("Unable to erase [%s]\n",result);
}
```

Return Value:

Unlink returns 0 if the file was successfully deleted or -1 if an error occurred and errno is set.

utime

Usage:

#include <time.h>
*int utime(char *path, time_t times[]);*

Description:

utime changes the modified time associated with the file specified by the path argument. The argument times contains the new time to be applied to the file. The times array should contain the last access time and a last modified time respectively. Since MSDOS has no concept of a last access time, this element is ignored. If the times argument is NULL, then the current time will be used.

Return Value:

utime returns a 0 on success and a -1 if an error occurred and errno is set.

See Also:

time

Example:

```
#include <time.h>
#include <stdio.h>
main(int argc,char *argv[])
{
   if(argc != 2)
   {
      fprintf(stderr," Usage:: utime [file.name]\n");
      abort();
   }
   if(utime(argv[1],NULL) == -1)
   {
      fprintf(stderr,"Could not update file [%s]\n",
                                          argv[1]);
      abort();
   }
}
```

va_arg - va_start

Usage:

```
#include < stdarg.h >
type va_arg(va_list arg_ptr, type);          /* ANSI */
void va_end(va_list arg_ptr);                /* ANSI */
void va_start(va_list arg_ptr, prev_parm);   /* ANSI */
```

Description:

These macros are used to maintain a list of arguments to be accessed within functions that accept a variable number of arguments (eg. the vprintf function).

va_list is a type of variable argument list defined in stdarg.h, together with the three macros, this allows variable argument lists to be processed by a function when it does not know the number of arguments being passed. The va_list array holds information required by va_arg and va_end. When a called function takes a variable argument list, it declares a variable of type va_list.

va_start is first called to initialize the argument list, for which arg_ptr points to the first argument in the va_list. The parameter prev_parm is the parameter preceding the first argument. After a call to va_start, a call to va_arg will evaluate data type from the location pointed to by arg_ptr and increment arg_ptr.

va_end resets arg_ptr to NULL.

Example:

See vfprintf.

Return Value:

va_arg returns the current argument.

va_start and va_end evaluate to void.

vprintf, vfprintf, vsprintf

Usage:

```
#include <stdio.h>
#include <stdarg.h>
int vfprintf(FILE *stream,const char *format, va_list arg_ptr);
                                                        /* ANSI */
int vprintf(const char *format,va_list arg_ptr);
                                                        /* ANSI */
int vsprintf(char *buffer,const char *format, va_list arg_ptr);
                                                        /* ANSI */
```

Description:

These functions are all similar to their counterparts printf(), fprintf() and sprintf() except the data is taken from the va_list arg_ptr. See counterparts for format.

Return Value:

The number of characters written if successful, or a negative number on error.

See Also:

fprintf, printf, sprintf

Example:

```
#include < stdarg.h >
#include <stdio.h>
varprint(char *format,...)
{
   va_list arg_ptr;
   va_start(arg_ptr,format);
   vprintf(format,arg_ptr);
   va_end(arg_ptr);
}
main()
{
   char *string = "STRING";
   int hnum = 0xb000;
   varprint("Call with a %s parm and a %xhex parm\n",
                                        string,hnum);
   varprint("Or with no parms\n");
}
```

write

Usage:

#include <io.h>
*int write(int fd,void *buffer,unsigned length);*

Description:

Write length bytes from buffer to the file specified by the file descriptor fd. This is a binary-only operation and is not buffered. Implemented as a direct DOS call.

Return Value:

Write returns the number of actual bytes written or a -1 if an error occurred and errno is set.

Example:

```
#include <io.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>
main()
{
   unsigned int fd;
   char *buffer = "Write this data to file";
   unsigned int count;
   int nwritten;
   count = strlen(buffer);
   fd = open("file.dat",O_WRONLY);
   if(fd == -1)
      {
         fputs("Unable to open file",stdout);
         return;
      }
   nwritten = write(fd,buffer,count);
   printf("\nWRITE:\n%u bytes written to file\n",
                                          nwritten);
   close(fd);
}
```

FLASH GRAPHICS INTRODUCTION

Flash Graphics is a set of graphics routines for the IBM PC (and clones) to interface to the Zortech C compilers. They are optimized for speed and size with most of the code being written in assembly language. Support is provided for Hercules, EGA (color and the monochrome modes), CGA, and VGA boards and Toshiba 3100.

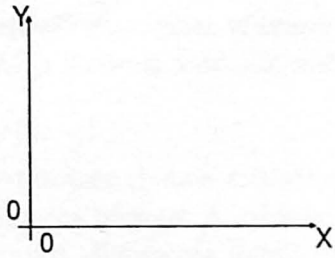
The initialization routine recognizes most graphics boards and automatically configures for the graphics board available. A manual over-ride is possible through either an environment variable or at compile time.

There are two sets of libraries available. One is a debug library that does range checking on most variables passed to the functions. If an error is detected then _assert () is called with an error message. This is to provide some help in finding (notoriously) difficult to locate program bugs. There is a small penalty paid in both size and speed with this library and so a non-debug version is availiable for use with fully debugged code.

The header file "fg.h" must be included in any modules using these functions and variables. Most of the function calls are really macros and will result in unresolved externals if the header file is not used.

Definition of Basic Coordinate System

FG uses a right - handed coordinate system.



Where x and y are pixel coordinates

Variable Types Defined

fg_coord_t Data type for all coordinates

fg_box_t Type for coordinate box. Boxes must be normalized before being passed to the graphics functions. A normalized box is one whose FG_X1 coordinate is less than or equal to the FG_X2 coordinate and FG_Y1 coordinate is less than or equal to the FG_Y2 coordinate. See Box and Line Indexes in the Constants Defined by FG section for more information. In other words, the first point of a box is always the lower left corner and the second is always the upper right corner.

fg_pbox_t Pointer to a box.

fg_line_t The type for a line. Lines use the same constants for indexes that boxes do, but have no need to be normalized before being passed to the functions.

fg_color_t The type for a color.

fg_pline_t Pointer to a line.

fg_handle_t A "handle" for a saved area of the screen. See fg_save() and fg_restore ().

Constants Defined by FG

Constants are indicated by upper case names. These are defined in the header file fg.h. They should always be referred to by the name and never the value that the macro in the header file expands to. They are subject to change in future versions of the graphics package.

Graphics Display Adapters

Values returned from fg_init_all() (see description of fg_init_all() for more information): The global fg_display is set to the same value.

Note: fg_init_all() will attempt to determine which graphics device is connected. This can be overridden at run time by using the environment variable FG_DISPLAY and setting it to one of the values below. E.g SET FG_DISPLAY=HERCFULL

FG_NULL	All graphics are routed to bit bucket
FG_HERCFULL	Hercules monochrome, 2 display pages
FG_HERCHALF	Hercules monochrome, 1 display page
FG_EGAECD	IBM EGA with enhanced color display
FG_EGACOLOR	IBM EGA with regular color display
FG_EGAMONO	IBM EGA with IBM monochrome monitor
FG_CGAHIRES	IBM CGA in 640x200 mode
FG_TOSHIBA	TOSHIBA 3100 mode.
FG_VGA11	IBM VGA in mode 0x11
FG_VGA12	IBM VGA in mode 0x12

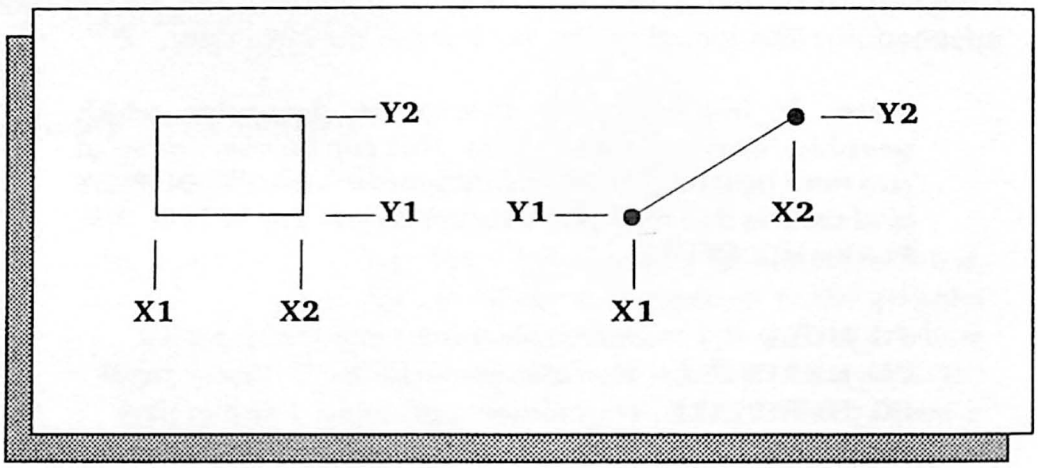
Other values for the environment variable FG_DISPLAY:

HERC	Hercules monochrome, fg_init_all to determine whether 1 or 2 pages

Box and Line Indexes

FG_X1
FG_Y1
FG_X2
FG_Y2

Used as indexes into boxes and lines, as in fg_displaybox [FG_X1] refers to the x coordinate of the left side of the display.



FLASH GRAHICS BOX/LINE COORDINATES

The above diagram shows the relationship of the FG_X? and FG_Y? indices to the coordinates of a box or line within Flash Graphics. These constants should be used to put data into the fg_box and line arrays rather than the absolute values to avoid problems with future display adapters which may use a different coordinate system.

Masks

Mask constants are passed to nearly all functions to indicate the bits to perform the desired operations on. The mask only has utility on boards that support more than two colors. For boards like the EGA which supports 16 simultaneous colors in the high resolution mode there are 4 bits in the mask that may be utilized by the programmer in some fashion. Whatever the board, the significant mask bits are always packed into the low order bits.

In most cases the application will want to write to all the bit planes. Using a value of ~0 will ensure this. Controlling which bit planes the functions operate on is useful for:

- *If you have 2 planes (2bits/pixel), you can draw a different image on each plane. By then changing the color map, the planes can be independently turned on or off, creating an animation effect.*

- *The cursor could be written to its own plane, thus making it independent of the other graphics.*

Writing Modes

One of the following is passed to nearly all functions to indicate the type of operation to perform on the screen.

FG_MODE_XOR	Mode value for XORing colors on to the screen.
FG_MODE_SET	Mode value for setting colors on to the screen.

TYPES OF LINES

One of the following is passed to the line drawing functions to indicate the type of line to draw.

FG_LINE_SOLID	
FG_LINE_LONG_DASH	
FG_LINE_MEDIUM_DOTTED	
FG_LINE_DASH_DOTTED	
FG_LINE_MEDIUM_DASHED	
FG_LINE_DASH_W_2_DOTS	
FG_LINE_SHORT_DASH	
FG_LINE_DENSE_DOTTED	
FG_LINE_SPARSE_DOTTED	
FG_LINE_USER_DEFINED	This line type is undefined and should be defined by the application program with fg_setlinepattern ().
FG_LINE_MAX	The maximum number of line types defined at one time.

Available Colors

This is the value that must be passed to functions as arguments for the color of lines, dots, boxes, etc. Due to the variety of graphics boards supported no one board supplies all the following colors. Some supply only two, FLASH GRAPHICS is defined such that there will allows be at least two of the same colors available on all boards supported. These values are what the defaults are. If you change the color map, you are on your own. If a color is not available, it will expand to -1. These values are not valid before fg_init() is called.

FG_BLACK	Always available
FG_BLUE	
FG_GREEN	
FG_CYAN	
FG_BLUE_GREEN	Blue-green - same as cyan.
FG_RED	
FG_MAGENTA	
FG_PURPLE	Purple - same as magenta
FG_YELLOW	
FG_WHITE	Always available
FG_GRAY	
FG_LIGHT_BLUE	
FG_LIGHT_GREEN	
FG_LIGHT_CYAN	
FG_LIGHT_BLUE_GREEN	Light blue-green = light cyan
FG_LIGHT_RED	
FG_LIGHT_MAGENTA	
FG_BROWN	
FG_LIGHT_WHITE	
FG_HIGHLIGHT	Normally an intense white
FG_BLINK	Blinking white
FG_COLOR_MAX	The max. no. of colors that can be supported by any device. See fg_ncolormap and fg_nsimulcolor.

PUBLIC VARIABLES

Values of these variables are garbage until fg_init( ) is called. They are garbage after fg_term() is called. The values (except fg_activepage and fg_displaypage) never change between fg_init() and fg_term()- none of them must ever be modified by the application.

fg_displaybox

Type:

fg_box_t fg_displaybox;

Description:

The bounding box of the display.

(fg_displaybox [FG_X1], fg_displaybox [FG_Y1]) is the coordinate of the lower left corner of the display.

(fg_displaybox [FG_X2], fg_displaybox [FG_Y2]) is the coordinate of the upper right corner of the display.

fg_charbox

Type:

fg_box_t fg_charbox;

Description:

The bounding box of a character.

(fg_charbox [FG_X1], fg_charbox [FG_Y1]) is (0, 0)

(fg_charbox [FG_X2], fg_charbox [FG_Y2]) is the coordinate of the upper right corner of the bounding box of the character.

fg_ncolormap

Type:

int fg_ncolormap;

Description:

The number of entries in the color map.

Example:

For the EGA board with an Enhanced Color Display this is 64. For the Hercules board this is 2.

fg_nsimulcolor

Type:

int fg_nsimulcolor;

Description:

The number of simultaneous colors possible, 2 raised to the nth power, where n is bits per pixel.

Example:

For the EGA board with an Enhanced Color Display this is 16 (4 bits per pixel). For the Hercules board this is 2 (1 bit per pixel). The number of bits.pixel is equivalent to the number of bit planes.

fg_pixelx, fg_pixely

Type:

int fg_pixelx, fg_pixely;

Description:

The typical width and height of pixels in micrometers. Since the actual monitor being used is impossible to determine with software, these numbers are for a typical monitor used with this graphics board.

fg_numpages

Type:

int fg_numpages;

Description:

Number of display pages, referred to by 0.. (fg_numpages -1).

Example:

The EGA board with 256k of RAM installed will have 2 different pages of graphics available which are referred to by the page numbers 0 and 1.

fg_display

Type:

int fg_display;

Description:

The type of graphics display board attached to the computer. This variable is set to one of the constants FG_NULL, FG_EGAECD, FG_HERCFULL, ... etc as described in the constants section.

fg_activepage

Type:

unsigned fg_activepage;

Description:

Current active page to which all graphics commands are directed. The active page is changed by the function fg_setactivepage ().

fg_displaypage

Type:

unsigned fg_displaypage;

Description:

The current page being displayed on the monitor. The display page is changed by the function fg_setdisplaypage().

OPTIONAL USER-SUPPLIED FUNCTIONS

The user of the fg module should supply the following functions although default versions are supplied in the libraries.

_assert

Type:

*void _assert(char *string_p, char *file_p, unsigned int line);*

Description:

_assert () is called when a program bug is detected such as a coordinate that is out of range and was not clipped. It should print out the message pointed to by "string_p", the source file name at "file_p" and the source file line number "line". It must not return, it is strongly suggested that it exit the program and return to DOS. It is also strongly suggested that fg_term be called before returning to DOS. This is to restore the graphics board to text mode.

fg_lineclip

Type:

int fg_lineclip(fg_pbox_t clip, fg_pline_t line_in,
fg_pline_t line_out);

Description:

fg_pbox_t clip The clipping box for the line line_in.

fg_pline_t line_in

The line that is to be tested against the clipping box.

fg_pline_t line_out
The output line that is a clipped version of line_in.

Returns non-zero if one or more pixels of the line "line_out" lie inside the clipping box "clip". Returns zero otherwise.

Clipping is an area prone to problems with roundoff and disconcerting behavior. If the programmer is not satisfied with the performance of the functions that call this routine (fg_drawlineclip (), fg_drawbox (), and fg_drawthickline ()) the programmer may supply an alternate.

FLASH GRAPHICS PUBLIC FUNCTIONS

fg_adjustxy

Usage:

```
#include <fg.h>
void fg_adjustxy (int rot, int n, fg_coord_t *px,
                  fg_coord_t *py, fg_pbox_t box);
```

Description:

int rot One of the constants FG_ROT0, FG_ROT90, FG_ROT180, FG_ROT270.

int n The number of chars or objects already output

fg_coord_t *px, *py
Pointers to the coordinates of the starting location of first character in the string.

fg_pbox_t box
The bounding box of the character or object.

The x or y coordinate for the next character (or object bounded by "box") is calculated depending on rotation. *px and *py are adjusted to be where the next character (object) would go. This is most useful with fg_drawmatrix () and fg_puts ().

Example:

```
{
    static char test_string[] = "This is a test.";
    fg_coord_t x, y;
    .
    .
    .
    x = fg_displaybox[FG_X1] + fg_displaybox[FG_X2];
    x -= sizeof (test_string) * (fg_charbox[FG_X2] + 1);
    x /= 2;
    y = (fg_displaybox[FG_Y1] + fg_displaybox[FG_Y2])/2;

    fg_puts (FG_WHITE, FG_MODE_SET, ~0, FG_ROT0, x, y,
                        test_string, fg_displaybox);

    fg_adjustxy(FG_ROT0, sizeof(test_string) - 1, &x,
                            &y, fg_charbox);
                    /* x and y now are located at the
                    end of the string "test string"*/
}
```

fg_blit

Usage:

#include <fg.h>
void fg_blit (fg_pbox_t src_box, fg_coord_t x, fg_coord_t y,
int dstpage, int srcpage);

Description:

Do a 'blit' of a rectangle of pixels from one area of the screen to another.

Input:

src_box	Defines the source rectangle of pixels. Box src_box must be enclosed by fg_displaybox both before the start of the blit and after the translation to the new coordinates.
x and y	Give the coordinates of the lower left corner of the destination of the box src_box.
dstpage	Which page the pixels are to be moved to.
srcpage	Which page the pixels are to be read from.

No clipping is done. Overlapping source and destination locations will be handled such that the pixel to be read is not destroyed by the write. This function can also be used as a "page copy" function with the proper src_box (fg_displaybox) and destination coordinates (0,0).

Example:

```
/* copy screen from oldpage to page */
fg_blit (fg_displaybox, 0, 0, page, oldpage);
```

fg_box_cpy

Usage:

#include <fg.h>
pbox_t fg_box_cpy (pbox_t to, pbox_t from);

Description:

Copy the box "from" to the box "to". Return pointer to the box "to".

Example:

```
fg_box_cpy (destination_box, source_box);
```

fg_drawarc

Usage:

#include <fg.h>
void fg_drawarc(fg_color_t color, int mode, int mask,
fg_coord_t x, fg_coord_t y, fg_coord_t radius, int startangle,
int endangle, fg_pbox_t clipbox);

Description:

Draw circular arc counter-clockwise from startangle to endangle.

Input:

x,y	Center of circle
radius	Radius of circle
clipbox	Bounding box of the arc.
startangle	Value from 0..3600 (10ths of a degree)
endangle	Value from 0..3600 (10ths of a degree). To draw a complete circle, use 0, 3600. Note that a zero radius or zero angular distance arc will be drawn as one pixel.

Although the units for the start and end angles are in 10ths of a degree, there is a maximum error of about 3 degrees for some angles. The arc is circular in pixel units. On most displays it will appear as an ellipse due to the difference in dimensions in the x and y directions of each pixel (the aspect ratio). If a circular appearance is required use fg_drawellipse () with the appropriate scaling for the x and y radii calculated with the use of fg_pixelx and fg_pixely.

Example:

```
fg_drawarc     (FG_WHITE, FG_MODE_XOR, ~0, 0,
        fg_displaybox [FG_Y2] + 10,
        fg_displaybox [FG_Y2]/2,
        1800, 3600, fg_displaybox);
```

fg_drawbox

Usage:

#include <fg.h>
void fg_drawbox(fg_color_t color, int mode, int mask,
int line_type, fg_pbox_t box,
fg_pbox_t clipbox);

Description:

Draw the outline of a box on the display screen and clip it to the box *clipbox*. The box to be drawn must be normalized, i.e. the first coordinate must be the lower left corner and the second must be the upper right corner.

Example:

```
{
    fg_box_t box_to_draw;
    box_to_draw [FG_X1] = -10;
    box_to_draw [FG_Y1] = 0;
    box_to_draw [FG_X2] = 20;
    box_to_draw [FG_Y2] = 30;
    fg_drawbox (FG_WHITE, FG_MODE_SET, ~0,
                    FG_LINE_DENSE_DOTTED,
                    box_to_draw, fg_displaybox);
}
```

fg_drawdot

Usage:

#include <fg.h>
void fg_drawdot(fg_color_t color, int mode, int mask,
fg_coord_t x, fg_coord_t y);

Description:

Draw dot at x,y. (x, y) must be within fg_displaybox.

Example:

```
fg_drawdot (FG_WHITE, FG_MODE_SET, ~0, 0, 0);
```

fg_drawellipse

Usage:

```
#include <fg.h>
void fg_drawellipse,(fg_color_t color, int mode, int mask,
                     fg_coord_t x, fg_coord_t y, fg_coord_t xradius,
                     fg_coord_t yradius, int startangle, int endangle,
                     fg_pbox_t clipbox);
```

Description:

Draw elliptical arc counter-clockwise from startangle to endangle.

Input:

x, y	Center of ellipse
xradius,yradius	Ellipse radii
clipbox	Bounding box for the ellipse.
startangle	Value from 0..3600 (10ths of a degree)
endangle	Value from 0..3600 (10ths of a degree)

To draw a complete ellipse, use 0 and 3600 for start and end angles respectively.

Note that a zero radius or zero angular distance arc will be drawn as one pixel.

See Also:

fg_drawarc

fg_drawline
fg_drawlinep

Usage:

#include <fg.h>
void fg_drawline(fg_color_t color, int mode, int mask,
int line_type, fg_pline_t line);
void fg_drawlinep(fg_color_t color, int mode, int mask,
int line_type, fg_pline_t line);

Description:

Draws line from (x1,y1) to (x2,y2) using indicated line type, color, mode and mask. The line must be clipped to fg_displaybox already. fg_drawlinep () differs from fg_drawline () in that the end points of the line will be drawn regardless of the line type.

Example:

```
{
   fg_line_t line_to_draw;
/* A line from the upper left corner to the lower */
                    /* right corner of the display. */
   line_to_draw [FG_X1] = fg_displaybox [FG_X2];
   line_to_draw [FG_Y1] = fg_displaybox [FG_Y1];
   line_to_draw [FG_X2] = fg_displaybox [FG_X1];
   line_to_draw [FG_Y2] = fg_displaybox [FG_Y2];
   fg_drawline (FG_WHITE, FG_MODE_SET, ~0,
                                FG_LINE_SOLID,
   line_to_draw);
}
```

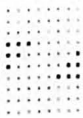
fg_drawlineclip

Usage:

#include <fg.h>
void fg_drawlineclip(fg_color_t color, int mode, int mask,
int line_type, fg_pline_t line, fg_pbox_t clipbox);

Description:

Clip the line to clipbox and draw the remainder. The function fg_lineclip () is used to do the clipping. If the results are not satisfactory the programmer may supply an alternate function with the same name (see "fg_lineclip ()"). See the description for fg_drawline () for more information on the input parameters.



fg_drawmatrix

Usage:

```
#include <fg.h>
void fg_drawmatrix(fg_color_t color, int mode, int mask,
                   int rotation, fg_coord_t x, fg_coord_t y,
                   char *p, fg_pbox_t b, fg_pbox_t clipbox);
```

Description:

Draw matrix of dots (b[X2]+1)*(b[Y2]+1) dots big.The matrix is drawn from top to bottom, i.e. the first row pointed to by p corresponds to the top row displayed. Only the bits that are 1s are output to the display. This is so that icons can be overlaid across what's already on screen. If a different result is desired you can do a fg_fillbox() prior to calling this routine.

Input:

color,mode,mask	The color, mode and mask to use for the matrix.
rotation	FG_ROTx about x,y where x,y is the lower left corner of where matrix is to be written.
p	pointer to data to be written, which is an array (b[X2]+1+7)/8 * (b[Y2]+1) bytes long. Bit 0 of the byte corresponds to the rightmost pixel, bit 7 to the leftmost.
b	Enclosing box. b[FG_X1] = b[FG_Y1] = 0.
clipbox	The clipping rectangle.

Example:

```
{
   static char mouse_cursor={0xF0,0xC0,0xA0,0x10,0x08};
   static fg_box_t mouse_box = {0, 0, 4, 3};
   fg_drawmatrix(FG_WHITE, FG_MODE_SET, ~0,
                 FG_ROT0, 0, 0, mouse_cursor,
                 mouse_box, fg_displaybox);


/* This is drawn as:

      X X X X 0
      X X 0 0 0
      X 0 X 0 0
      0 0 0 X 0
      0 0 0 0 X

Where the X's are set to WHITE and the 0's are
unchanged. */
}
```

fg_drawthickline

Usage:

```
#include <fg.h>
void fg_drawthickline(fg_color_t color, int mode, int mask,
                      int line_type, fg_pline_t line, fg_pbox_t clip,
                      int thickness);
```

Description:

Draw thick line. Line is drawn as if the 'pen' is a square that is 'thickness' pixels wide on a side. See fg_drawlineclip () for a description of the other parameters and a (similar) example.

fg_fillbox

Usage:

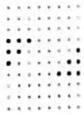
#include <fg.h>
void fg_fillbox(fg_color_t color, int mode, int mask, fg_pbox_t box);

Description:

Set box to color. Box b must be already clipped. This function can be used to clear the screen if the box is equal to fg_displaybox.

Example:

```
/* clear the screen */
fg_fillbox (FG_BLACK, MODE_SET, ~0, fg_displaybox);
```

fg_flush

Usage:

#include <fg.h>
void fg_flush (void);

Description:

Flush any pending output to display. Wait until the display is up-to-date. This should be called before any user input is expected to ensure that the screen is up-to-date.

Example:

```
fg_flush ();
```

fg_getcolormap

Usage:

#include <fg.h>
*void fg_getcolormap(fg_color_t *color_array, int max_entries)*

Description:

Get values of the current color map (inverse of fg_setcolormap()).

Input:

max entries Max # of entries to write into color_array[]. The actual number of entries written is the lesser of (fg_nsimulcolor or max) entries.

Output:

color_array Array of colors. The values are between 0 and fg_ncolormap - 1.

Example:

```
{
    fg_color_t color_array [16];
    fg_getcolormap (color_array, 16);
}
```

fg_init

Usage:

#include <fg.h>
void fg_init (void);

Description:

Initialize the graphics mode after fg_term has been called. See fg_init_all () and siblings. fg_init will restore the graphics mode in effect before the last fg_term() was called.

Example:

```
fg_init ();
```

This is useful if you need to temporarily switch to text mode:

```
fg_init_all();

/* MORE CODE*/

fg_term();
system(."dir");
fg_init();

/* MORE CODE */

fg_term();
```

fg_init_null - fg_init_vga12

Usage:

#include <fg.h>

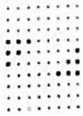
Function Prototype	Environment Variable for fg_init_all
int fg_init_null(void);	
int fg_init_all(void);	
int fg_init_egaecd(void);	*FG_EGAECD*
int fg_init_egacolor(void);	*FG_EGACOLOR*
int fg_init_egamono(void);	*FG_EGAMONO*
int fg_init_herc(void);	*FG_HERC or FG_HERCFULL*
int fg_init_vga11(void);	*FG_VGA11*
int fg_init_vga12(void);	*FG_VGA12*
int fg_init_toshiba(void);	*FG_TOSHIBA*
int fg_init_cga(void)	*FG_CGAHIRES*

Description:

Open Display Device: If not already opened save current mode of display device, so it can be restored by fg_term(). Set display device into its graphics mode. Set the current display page to page 0. Store the characteristics of the display device into the global variables.

Output: fg_display is set to the return value, fg_activepage and fg_displaypage are set to 0. Page 0 is set to FG_BLACK.

Returns: FG_XXXX Display type that was opened. FG_NULL if failure.



fg_init_all() will attempt to determine the type of display graphics that is available and open that display device. The environment variable FG_DISPLAY=XXXXX will override. The environment variable override will be primarilly useful for oddball clone boards, or for boards that emulate multiple other boards.

fg_init_herc() will assume that the graphics device is a Hercules board. It will determine if there are 1 or 2 pages available.

fg_init_null() is the bit-bucket graphics device. It always returns FG_null.

The first attempt to open the graphics device MUST be one of the *fg_init_xxx()* functions. Subsequent attempts to open the display (after changing back to text mode with *fg_term()*) may be made using *fg_init()*. Calling *fg_init()* will be much faster than calling *fg_init_xxxx()*. As new devices are implemented, corresponding functions will be named and available.

Example:

```
{
    if (fg_init_all () == FG_NULL)
    {
        fputs ("Unable to open graphics device.\n",
                                                stderr);
        exit (1);
    }
}
```

fg_line_cpy

Usage:

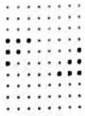
#include <fg.h>
fg_pline_t fg_line_cpy (fg_pline_t to,fg_ pline_t from);

Description:

Copy the line "from" to the line "to". Return "to".

Example:

```
fg_line_cpy (destination_line, source_line);
```

fg_putc

Usage:

#include <fg.h>
void fg_putc(fg_color_t color, int mode, int mask,
int rotation, fg_coord_t x, fg_coord_t y, char out_char,
fg_pbox_t clip_box);

Description:

Send character to screen using rotation, color, mask and mode. x, y gives the position of lower left corner of the character (about which point the character rotates). The character pixels are clipped to clipbox.

Example:

```
fg_putc (FG_WHITE, FG_MODE_SET, ~0,
         FG_ROT270, 100, 100, 'Z',
         fg_displaybox);
```

fg_puts

Usage:

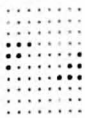
#include <fg.h>
void fg_puts(fg_color_t color, int mode, int mask,
int rotation, coord_t x, coord_t y,
*char *out_string, fg_pbox_t clip);*

Description:

Output a string to the graphics device. Use the specified color, mode, mask, and rotation. The starting point of the lower left corner of the first character is at x, y. The string to output is out_string, the string will be clipped to the box clip.

Example:

```
fg_puts (FG_WHITE, FG_MODE_SET, ~0, FG_ROTO, 0, 0,
         "Hello world.", fg_displaybox);
```

fg_readbox

Usage:

#include <fg.h>
*void fg_readbox (fg_pbox_t b, fg_color_t *p);*

Description:

Read box b (inclusive) pixels off of the screen and store into the fg_color_t array pointed to by p. The color of each pixel is written into each element. The number of bytes written to is therefore equal to the number of pixels enclosed by b multiplied by sizeof (fg_color_t). Rows are written first, in order of increasing y (this means starting at the lower left corner).

Example:

```
{
    fg_color_t *color_p;
    fg_box_t read_box;
    unsigned int bytes;
    read_box [FG_X1] = 10;
    read_box [FG_Y1] = 0;
    read_box [FG_X2] = 100;
    read_box [FG_Y2] = 100;
    bytes = read_box [FG_X2] - read_box [FG_X1] + 1;
    bytes *= read_box [FG_Y2] - read_box [FG_Y1] + 1;
    bytes *= sizeof (fg_color_t);
    color_p = malloc (bytes);
    assert (color_p != NULL);
    fg_readbox (read_box, color_p);
    .
    .
    .
                        /* Need not be the same box,
                       but must be the same size. */
    read_box [FG_X1] += 20;
    read_box [FG_X2] += 20;
    read_box [FG_Y1] += 30;
    read_box [FG_Y2] += 30;
    fg_writebox (read_box, color_p);
}
```

fg_readdot

Usage:

#include <fg.h>
fg_color_t fg_readdot(fg_coord_t x,fg_coord_t y);

Description:

Read dot at x,y.

Example:

```
{
   fg_color_t dot_color;
   dot_color = fg_readdot (0, 0);
}
```

Return Value:

Returns the color at x,y.(x,y) must be within fg_displaybox.

fg_restore

Usage:

#include <fg.h>
void fg_restore (fg_handle_t handle);

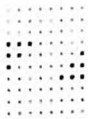
Description:

Restore an area to the screen that was saved by fg_save().

Input:	handle	Value returned by fg_save().
Output:	handle	is no longer valid. It may not be used on subsequent calls to fg_restore().

Example:

See fg_save ().



fg_save

Usage:

#include <fg.h>
fg_handle_t fg_save (fg_pbox_t box);

Description:

Save an area on the screen. This area can be restored using fg_restore(). fg_save() and fg_restore() are primarilly useful to support pop-in areas on the screen. Note that areas saved can only be restored to the same location (or the same location on another page if the active page was changed).

Example:

```
{
   fg_box_t save_box;
   fg_handle_t save_handle;
   save_box [FG_X1] = save_box [FG_Y1] = 0;
   save_box [FG_X1] = save_box [FG_Y1] = 100;
   save_handle = fg_save (save_box);
   assert (save_handle != NULL);
   .
   .
   .
   .
   fg_restore (save_handle);
}
```

Return Value:

fg_handle_t	Handle to be used with fg_restore()
NULL	Failed (probably out of memory)

fg_setactivepage

Usage:

#include <fg.h>
void fg_setactivepage (unsigned int pagenum);

Description:

Change active page to pagenum. The active page is the page that all the fg_xxxx routines draw to. It is not necessarilly the one that is being displayed. pagenum must be less than fg_numpages.

Output: fg_activepage is set to pagenum.

Example:

```
fg_setactivepage (1);
```

See Also:

fg_setdisplaypage

fg_setcolormap

Usage:

#include <fg.h>
void fg_setcolormap (fg_color_t carray[], int max);

Description:

Set all the colors for the color map. The color map is not modifiable if fg_ncolormap == fg_nsimulcolor.

Input:

carray Array of colors, dimension is fg_nsimulcolor. The values are between 0 and fg_ncolormap-1.

max Max number of entries to read from color_array[]. The actual number of entries read is the lesser of (fg_nsimulcolor or max).

fg_setdisplaypage

Usage:

#include <fg.h>
void fg_setdisplaypage(unsigned pagenum);

Description:

Change display page to pagenum. The display page is the page that the user sees on the screen. pagenum must be < fg_numpages.

Example:

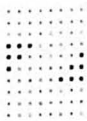
```
fg_setactivepage (0);
```

See Also:

fg_setactivepage

Output:

fg_displaypage is set to pagenum



fg_setlinepattern

Usage:

#include <fg.h>
void fg_setlinepattern(int line_type,int pattern);

Description:

Set the pattern for a line.

Input:

line_type	Value between 0 and FG_LINE_MAX-1. Subsequent uses of this value will use this line pattern.
pattern	16 bits defining the line pattern. The pattern starts at bit 15, goes down to bit 0, and then repeats. If fg_drawlinep() is to always generate a dot at the beginning of a line, bit 15 must be a 1.

Example:

```
{

   /* A very, very sparse dotted line. */
   fg_setlinepattern (FG_LINE_USER_DEFINED, 0x8000);
}
```

fg_term

Usage:

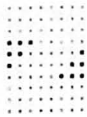
#include <fg.h>
void fg_term(void);

Description:

Close display device. Display is returned to pre-fg_init() state. Do nothing if fg_init() was never called, or fg_term() was already called without another intervening fg_init(). This is so that panic-and-abort routines can reset back to text mode, without regard to whether the display is in graphics mode or not.

Example:

```
fg_term ();                    /* Back to text mode. */
```

fg_writebox

Usage:

#include <fg.h>
*void fg_writebox (fg_pbox_t box, fg_color_t *p);*

Description:

Inverse of fg_readbox(). Writes the pixels to the screen.

Input:

box The bounding box of where the data is to be written.

p A pointer to an array of colors, one color for each pixel in the bounding box "box".

Example:

See fg_readbox

See Also:

fg_readbox

PART FOUR

the appendices

APPENDIX ONE

ZorLib

The Zortech Librarian

ZORLIB is the Zortech librarian utility. It is compatible with the Microsoft linker and librarian programs, as well as with the Zortech linker. Libraries can be created, and object files can be added, removed or substituted.

What is a librarian?

A librarian is a utility which maintains object library (.lib) files. Examples of such files are the runtime libraries supplied with this compiler. If you have purchased the optional C and C++ library source code you can use ZORLIB to customize these runtime libraries to your own requirements. With the aid of ZORLIB you can also create and maintain your own libraries of useful functions without resorting to large numbers of *#include* directives in you programs. Libraries are useful in collecting together a set of related object files, for example graphics functions, which can then be brought to the linker as a single convenient file, rather than as many separate ones. No change is made to the content of the object files, or to how they work: a library file just gathers them together in one place.

With the aid of ZORLIB libraries can be created and objects inserted, removed or updated. Copies of objects within a library can be created as separate external object (.obj) files. ZORLIB will also generate a readable listing of the contents of a library.

ZORLIB is intended to make the best use of the large amounts of memory available in modern PC's and is very fast, but the it's design makes it unsuitable for handling very large libraries ie. those larger than 500K.

Using ZORLIB

ZORLIB can be invoked on the command line, in a batch file, or by MAKE. There are three ways of passing arguments to ZORLIB, on the command line, via a response file similar to a linker response file, or through an interactive mode similar to that available with the Microsoft librarian.

The Zorlib command line has the following format:

```
[d:][path]ZORLIB lib-file[switches][,action-files][,list-file][;]
```

where: ([] indicates optional parameters)

[d:] and [path] are the disk drive and path on which DOS can find ZORLIB

lib-file is the filename of the library file to be created or amended. If no extension is given the extension .lib is assumed.

[switches] can be one or more of the following:

/C Create

If the library does not already exist then you must include this switch to confirm that you want a new library to be created.

/N No backup

Normally when ZORLIB updates a library file it will save the old file as a backup file (.bak). This is sometimes valuable when you regret a change, but extra disk space is required, and this may not always be available especially on systems which do not have a hard disc drive. If you do not wish to keep backup versions then specify the /N switch.

/B Both cases

Upper- and lower-case letters will be treated as distinct. This applies only to public symbols. Names for modules are always converted to upper case since they must be interchangeable with MSDOS file names.

[action-files] These are one or more object files, which must in the current directory . If no extension is supplied then .obj will be assumed. When the objects are placed in the library, Zorlib will create a module name equal to that of the object file (without the extension).

At least one of the characters -, + or * must appear before each object file name. In some cases two of these characters can be used. Their meanings are:

- Remove the named object from the library.

  This will cause an error message and be ignored if the object is not in the library.

+ Add the named object to the library.

  This will cause an error message and be ignored if an object of that name is already present, or if the named file cannot be found

* Make a copy of the object as a separate .obj file outside of the library.

  This will cause an error message and be ignored if the object is not in the library.

Some pairs are legal:

+- or -+ This will replace an existing object with a new version, the - operation being performed first.

-* or *- This will remove an object from the library while at the same time making a separate external copy of it as an .obj file. The * operation is always performed first.

[list-file] The optional list file name may by any MSDOS pathname. If no extension is given then .lst is taken as the default. This file will be produced by ZORLIB and contains a list of all of the object files which are in the library together with the public symbols that they provide. If no list file name is given, then no listing will be generated. If a list file is named, then the listing will be generated even if there are no actions specified.

Using Zorlib with Interactive Prompts.

If you invoke ZORLIB with no arguments, the program will enter an interactive mode, with the user being prompted for the required input. The & character can be used to continue input onto a new line if required.

Using Response Files.

ZORLIB can also be invoked with the address of a response file. This is a file which contains the responses that the compiler would normally require if it were in interactive mode. An example response file could contain :

```
\ZORTECH\LIB\MYWINDOW.LIB /C
+WINOPEN +WINBLANK +WINMOVE +WINHIDE &
+WINSHOW +WINPUTS +WINGETS +WINPRINT
\ZORTECH\MYWIN.LST;
```

note the use of the & symbol to continue on another line.

If the response file were called makewin.rsp, then the librarian is invoked thus:

```
ZORLIB @MAKEWIN.RSP
```

Note that you must not have any filename beginning with a @ when working with ZORLIB, because it will be mistaken for a reference to a response file.

Capacity

ZORLIB is designed to run fast on modern large memory PC's, and works best on equipment with 512K or more memory. ZORLIB requires approximately 50K for itself, and holds whole files in memory in order to maintain high performance. Therefore the maximum library size that ZORLIB can handle is determined by the available free memory.

In any case ZORLIB has a practical limit of 500K on the size of library which can be manipulated.

Notes

When adding an object file to the library ZORLIB uses the file name converted to upper case and with its extension removed to form the module name in the library.

ZORLIB constructs a symbol table and will not allow duplicate public symbols.

ZORLIB assumes that any file not ending in .lib is an object file. Libraries can be concatenated provided the library being added has the .lib extension. ZORLIB does not change the module names from that file.

Empty

APPENDIX TWO

Compiler Error Messages

ZTCPP1, ZTC1 and ZTC1B

For errors in the source file, the compiler will usually print out the offending line with an ^ underneath the point at which the error was detected. This is not necessarily where the real error is, it is only where the parser was when the error was detected. If the ^ is under a macro, and a confusing error message is printed, try compiling with the -e switch. This will show the effect of the macro processor and may help to pinpoint the actual error.

Example:

```
A = B;
^
"JUNK.C" line 15:syntax error: undefined identifier 'A'
```

The line in which the error was detected is displayed followed by the error message. The error message contains the file name and line number of the erroneous line. The type of error (warning, syntax error, error, or fatal) is shown followed by the error description.

Some of the following error descriptions contain specific data about the error. For example, the error description above contains the name of the undefined identifier. In the following error description <IDENT> is used to show where a user-named item will be placed, <NUMB> where a user-defined number is placed, and <filename> where a file name is placed.

C++ Extension Errors

trailing parameters must have initializers
Parameters with default initializers must occur at the end of the parameter list.

bit field <IDENT> can't be in an anonymous union
You cannot use a bit field in an anonymous union.

members are already public
you have declared members public which are already so.

not possible to derive from union <IDENT>
A union can't be a base class.

static variables in inline functions not allowed
you may not use static variables in an inline function.

type is too complex
Identifier with type info appended exceeds 127 chars, which is the max length of an identifier.

cannot uniquely determine which function <IDENT> to call
Function <IDENT>is overloaded, and the compiler can't determine from the argument types which function is intended. This message also appears when the compiler is looking for a constructor with no arguments or a constructor with one reference argument (the copy initializer).

not an overloadable operator token
Not all operators can be overloaded.

operator overload must be a function

incorrect number of parameters for operator

Operator overloaded functions must be unary, binary or n-ary depending on which operator is overloaded. () is n-ary.

at least one parameter must be a class or a class&

Operator overloaded functions that aren't class members must have at least one parameter being a class or class&.

<IDENT> must be a class name preceding '::'

<IDENT> must be the base class

<IDENT> must be in public section

illegal member declaration

Used as a catch-all for miscellaneous errors in defining class members, such as not declaring destructors right.

must be in a member func to call a member func '::' expected

You may only call a private member function from another member function of the same class.

type conversions must be members

A user-defined type conversion function was declared outside of a class.

statics with constructors or destructors must have file scope

and

already seen base initializer

A static that is initialized with a constructor or destroyed with a destructor cannot be local to a function. It must be declared outside of any function.

bad member-initializer

Possibilities are:

1. The member-initializer syntax is not recognized.

2. More than one member-initializer appears for member

3. A base class initializer appears for class but there is no base class.

vectors cannot have initializers

You cannot initialize a vector of objects aith a constructor that has an arqument list.

can only delete pointers

You have tried to use the delete keyword on a data type other than a pointer.

storage class for <IDENT> can't be both extern and inline

You cannot use the inline type specifier for a function declared external.

operator functions () and [] must be members

Operator functions () and [] can only be used within a class definition.

one argument required for member initializer for <IDENT>

For member initializers for which the member has no constructor there must be exactly 1 parameter. The member is initialized by assignment.

Standard C Errors

'#else' or '#elif' found without '#if'

There are more '#else' or '#elif' than preceding '#if', '#ifdef' or '#ifndef's.

'#endif' found without '#if'

There are more '#endif' than preceding '#if', '#ifdef' or '#ifndef's.

' < NUMB > ' actual arguments expected

A macro was defined with an argument that was not present when the macro was used.

' < IDENT > ' is already defined

The item was declared previously.

' < IDENT > ' is not a correct struct, union, or enum tag identifier

The structure, union, or enum has already been defined.

' < IDENT > ' is not a member of this struct or union

The member of the struct/union reference is not a member of this structure. This can be overridden using a cast. This message could be encountered when compiling code developed on a pre-Version 7 compiler.

' < IDENT > ' is not a variable

The identifier has not been declared as a variable.

' < IDENT > ' is not in function parameter list

The named parameter was not listed as a parameter to the function.

' < IDENT > ' previously declared as something else

The identifier was previously declared as another type. This may be an automatically declared function now being declared as returning something other than an int.

' < IDENT > ' storage class is not allowed in this context

The auto and register storage classes cannot be used outside functions, extern cannot be used for function arguments, etc.

' < IDENT > ' unrecognized parameter

The specified command line parameter is not supported by this compiler.

'(' expected

The expression following an 'if', 'while', and 'for' must be
enclosed in '( )'.

')' expected

The ')' token is required here. This can be caused by mismatched '( )', a bad expression, or other errors.

':' expected

A colon is expected following the constant expression in a case statement.

';' expected

The end of an expression was found and requires the semi-colon.

'=', ';' or ',' expected

The previous declaration was not properly formed. The end of the declaration was encounted none of the three listed was found.

']' expected

The array declaration or usage was not ended with a ']'.

'default:' is already used

Two default statements were found in a single switch statement. The concluding 'while'

'while' expected
The concluding 'while' was expected for the do-while statement.

'{' expected
The C language syntax requires an '{' here.

'{' or tag identifier expected
A tagname or a { is required after a struct union or enum keyword.

'}' expected
The C++ language syntax requires an "}" here.

array index is long
Only int subscripts make sense, as only a 16-bit offset is possible.

array of functions is not allowed
This is an illegal type definition. You may want an array of pointers to functions. Example: int (* x []) ( ).

bad -D switch, < IDENT >
The command line macro definition was illegal.

bad abstract_declarator
An invalid abstract declarator was encountered.

bad file name ' < FILENAME > '
The file name is illegal and cannot be used.

blank arguments are not allowed
Macro arguments must be defined.

'break' is valid only in loop or switch
A break statement can only be used in the for, while, switch, or do-while statements.

can't build filespec ' < FILENAME > '
The named file could not be written. This most likely means a full disk.

can't init auto structs or auto arrays
An attempt was made to initialize an automatic aggregate which is not legal in C++ .

can't nest comments
Due to the very obscure bugs that can result from this, the authors have decided to make attempts to nest comments a syntax error. Use #if 0 and #endif to comment out sections of code.

cannot define parameter as extern
Extern is an illegal storage class for a function parameter.

case < NUMB > was already used
The value shown has already been used within the switch statement.

constant initializer expected
Initializers must be a constant or the address of a previously declared static item.

'continue' is valid only in loop or switch
A continue statement was used out of context. It can only be used in the for, while, or a do- while statement.

ZTC1, ZTCPP1 bug
There is a bug in the compiler. Please notify ZORTECH.

ZTC2 bug
There is a bug in the compiler. Please notify ZORTECH.

divide by 0
A constant expression involving a divide by zero was encountered. Check any macros.

duplicate file names ' < FILENAME > '
Cannot open two files at once with the same name.

end of file found before '#endif'
The end of file was reached in the middle of a conditional compilation statement list. Add an #endif.

end of file found before end of argument
The compiler was trying to find the end of macro argument when the end of file was encountered.

end of file found before end of comment
The end of file was reached in a comment. Missing a '*/'.

end of file found before end of macro definition
The macro definition may be missing a ')' or it may be in a quote.

end of file found before end of string
An end of file was encountered within a string. This can be caused by a missing " .

end of line expected
Extra characters were found on a line and were not needed.

error writing output file
Usually caused by a full disk.

expected data def of ' < IDENT > ', not func def
The item was declared to be a function but cannot be, due to its storage class or scope.

exponent expected
The exponent must follow the 'e' without spaces in between.

expression expected
The syntax requires an expression at this point. A ; or } may be missing.

field ' < IDENT > ' must be int or unsigned
Bit field structure members must be of type integer or unsigned.

filespec string expected
A file name is expected in the double quotes or angle brackets following an #include command.

function expected
A function declaration expected and not found. This could be caused by mismatched { }.

functions can't return arrays or funcs
The function was declared to be return an object that cannot be returned.

generic error
A compiler error. Please contact ZORTECH.

hex digit expected following '0x'
A hex number is required without any leading spaces.

identifier expected
An identifier is required here but another token was found.

identifier is too long
An identifier is limited to 127 characters, any more is considered an error.

identifier or '(declarator)' expected
A declaration of a static or extern variable or function was expected. If this occurred in a function then there are more left braces '{' than right braces '}'.

illegal cast
The item being cast cannot take on the cast type.

illegal character, ASCII < NUMB > decimal
The character is not a token by itself and cannot be used within a multi-character token.

illegal operand types
The operand types cannot be used with this operator. A cast may help.

illegal parameter declaration
The parameter declaration was improperly formed.

illegal pointer arithmetic
Integers may be added or subtracted from pointers; also two pointers may be subtracted or compared. Other operations on pointers are illegal.

illegal type for ' < IDENT > ' member
The member must be unsigned.

integer constant expression expected
Integer constants are required after case statements, for array size declaration, and the #if, #elif, #exit, and #line preprocessor commands.

integer expression expected
Integer expressions are required for array indices and switch expressions.

integer is too large
The constant number is larger than can be represented with a 16-bit integer.

intermediate file error: TG < NUMB >
A compiler bug. Please notify ZORTECH.

line number expected
A constant expression for a line number is required.

lvalue expected
The expression must be a variable that can be assigned a value. The variable may be declared const.

macro ' < IDENT > ' is already in use
The named macro is circularly defined. Example: #define UNIX UNIX

macro ' < IDENT > ' is not defined
An attempt was made to undefine a macro that was not defined.

maximum of < NUMB > include paths was exceeded
There are more include paths than the compiler allows.

maximum of < NUMB > nested conditionals exceeded
The limit was reached on nesting of #if.

maximum width of < NUMB > bits exceeded
There can only be 16 bits in a bit field.

memory overflow
The compiler ran out of memory! Memory is used up mainly by the symbol table and macro definitions. Lots of variables use up space too. The compiler is limited to 64kb of data, so having more than 128kb for the compiler will not help. The work around is to break up the function being compiled or the file being compiled.

must specify array index
The attempted array declaration must contain the second and all following indices.

no input file specified
An input file is required on the command line.

not a struct or union type
The type of object preceding the . or -> is not a struct or union.

not in a switch statement
A 'case' or 'default' statement was encountered outside of a switch statement.

number <NUMB > is too large
The number is larger than can be represented with a long type.

parameter list is out of context
Parameters were included in a function definition that are illegal and discarded. Example:
extern junk (ab, cd);

pointer mismatch
The pointers point to different kinds of objects. Could indicate an error in the program. This can be suppressed using a cast.

possible unintended assignment
A "=" was found in an "if()" or "while()" where a "==" may be desired. This may or may not be an error. The warning can be stopped by rewriting the statement as follows:
if(x=y) to if((x=y) != 0).

premature end of source file
The end of file was encountered in a function.

size of ' < IDENT > ' member is not known
A reference is being made to struct or array with an undefined size.

string expected
A string with a filename is expected after a #include.

struct or union is forward referenced
A reference was made to a structure or union before it was defined.

too many errors
The compiler error limit of 4 was reached. The error limit maximum can be disabled using the "-x" option.

too many initializers
There were too many initialize elements for the item. Example:
int a[3] = { 0, 1, 2, 3, 4, 5};

unable to open input file ' < FILENAME > '
The input file was not found on the specified disk.

unable to open output file ' < FILENAME > '
The file name could be invalid or the disk could be full.

undefined identifier ' < IDENT > '
The identifier was not previously defined.

undefined label ' < IDENT > '
A goto to the label listed was encountered, but the label was not defined within the function.

undefined tag ' < IDENT > '
The structure or union is not defined.

undefined use of struct or union
A structure or union was illegally used. Example: adding two structures.

unrecognized parameter ' < IDENT > '
This command line option is not used by the compiler.

unrecognized pragma '# < IDENT > '
The specified preprocessor command is not supported with this compiler.

unrecognized token
The token encountered is not part of the language.

Link Error Messages

Warning: LINK does not support the Huge memory model.

Warning: COMMENT record with unknown memory model '<IDENT>'

Error: unknown memory model '<IDENT>' These error messages could be caused by linking in .OBJ files produced by other compilers.

Error: file <IDENT> was compiled with memory model <IDENT>, but other files were compiled with memory model <IDENT>. LINK checks the memory model that each .OBJ file was compiled with, to ensure that they match.

Error: no .EXE file specified

Warning: option '/<IDENT>' is ignored

Error: option '/<IDENT>' is invalid

Error: '<IDENT>' is an invalid number An invalid or non-supported link switch was entered on the command line.

Error: more than one _CTDT_PTR table has been found

Warning: <NUMB> static constructors and destructors : but the table holds only <NUMB>

Error: static constructors and destructors were found but no _CTDT_PTR

Error: unable to determine pointer size of _CTDT_PTR table

Error: offset for <IDENT> in _CTDT_PTR table is too large errors involving C++ extensions.

Error: error writing .EXE file. Could be out of disk space.

Error: duplicate entry points More than one start address was found.

Warning: no stack segment Ignore this message if creating a .COM file.

Warning: stack size specified with /STACK: option is smaller than the : stack segment (<NUMB> versus <NUMB>).

Error: no start address, default to 0x100

Warning: target out of frame in self-relative fixup

Warning: location out of frame in self-relative fixup

Warning: target out of frame

Error: self-relative fixup out-of-range, byte(%ld)

Error: self-relative fixup type is invalid

Error: target in segment-relative fixup out of frame

These errors can be caused by defining an identifier as data in one module and as a function in another.

Error: can't find library file '<IDENT>'

Error: '<IDENT>' is not a library file

Error: <IDENT> is defined more than once in file <IDENT> (module <IDENT>)

Error: <IDENT> is a FAR COMDEF with unequal element sizes <NUMB> and <NUMB>

Error: undefined symbols:

Error: can't find object file '<IDENT>'

Error: '<IDENT>' is not an object file

Error: FAR COMDEF <IDENT> exceeds 64k by <NUMB> bytes

Error: NEAR COMDEFs exceed 64k by <NUMB> bytes

Error: segment <IDENT> is greater than 64k (by %ld bytes) Reduce the size of the segment or compile with the large code model.

Error: can't open "<IDENT>" , Can't open file.

Technical Information

The errors below involve invalid .OBJ records. They are documented mainly for people who understand .OBJ records.

ObjCheck:	SEGDEF.A=<NUMB> not handled
ObjCheck:	SEGDEF.C=<NUMB> not handled
ObjCheck:	SEGDEF.B, big segments not handled
ObjCheck:	SEGDEF.P, page resident segments not handled
Error:	SEGDEF.segidx is out-of-range
Error:	SEGDEF.classidx is out-of-range
ObjError:	GRPDEF.group index is out-of-range
ObjError:	GRPDEF, not SI record
ObjError:	GRPDEF.seg index out-of-range
Error:	PUBDEF.BASE SEG index out of range <NUMB> > <NUMB>
Error:	PUBDEF.BASE GRP index out of range
Error:	LEDATA.SEG index out of range
Error:	LIDATA.SEG index out of range
ObjError:	FIXUPP isn't a thread or a fixup
ObjError:	FIXUPP.S, 3-byte offset not supported

ObjError: FIXUPP.LOC, illegal value

ObjError: MODEND.L, physical address not supported

ObjError: only 1 and 2 byte COMDEF lengths are supported

Error: LINNUM.GRP index not zero

Error: LINNUM.SEG index is zero

Error: LINNUM.SEG index out of range

Error: corrupt record in object module in '<IDENT>'

Error: premature end-of-file in object module in '<IDENT>'

Error: unsupported object record, type=<NUMB> in '<IDENT>'

Error: unknown object record, type=<NUMB> in '<IDENT>'

Error: self-relative fixup type is invalid

Zorlib Error Messages

Warning messages

<IDENT> already exists, not changed

You are trying to add a module, but a module of that name already exists in the library and you did not remove it first, If you are updating with a new version of the same module use the '-+' action.

<IDENT> not found, can't delete or extract it

Zorlib could not find any module with the name you specified. Get Zorlib to produce a list file to check the contents of the library.

public <IDENT> already in library, this module skipped

It is not allowed to have to modules which contain the same public symbol. Zorlib will refuse to add the second module to the library.

unknown switch <IDENT> in command line, ignored

Valid switches are /C (create new library), /N (no .bak file) and /B (both cases distinct).

Error Messages

<IDENT> - path is too long

MSDOS pathnames have a maximum of 63 characters

<IDENT> does not exist: use /C to create

ZORLIB was unable to find the specified library file, and the /C flag was not used. You must use the /C switch on the command line if you wish to create a new library

'@' seen, expected a response-files name

The '@' symbol is used as a switch followed by a file name in which the rest of the command line is to be found. ZORLIB has seen the '@' but you did not provide a file name.

a library file must be named

You have called up ZORLIB but did not specify a library file. The library file name must be the first parameter on the command line or the response to the first prompt.

create requested but <IDENT> already exists

You have used the /C switch but the name chosen for the new library already exists, rename or delete the old library, or choose a new name.

the combinations '+*' or '*+' are not allowed

ZORLIB has no ability to add and extract in one operation. You may have mistyped.

unexpected char <IDENT> in command line

That character is not recognised by ZORLIB or is in the wrong position.

System Errors

ZORLIB will also report MSDOS errors which occur during operation, such as memory full or file read/write errors.

Make Error Messages

When MAKE encounters an error, a message is printed on the screen describing the error and MAKE returns to DOS. The error messages are listed below with a description of each.

--- errorlevel nnn

When a rule is executed and it returns a non-zero exit status, this message is printed and MAKE terminates (unless MAKE was instructed to ignore the exit status).

undefined switch 'xxxx'

bad macro definition 'xxxx'

can't read makefile 'xxxx'

target must appear before commands
Can't specify rules before any targets.

unrecognized target 'xxx'
Target was not one of .SILENT or .IGNORE.

command line too long
Rules to be executed by COMMAND.COM cannot exceed 124 bytes in length.

expanded line is too long
Argument to a command exceeds 127 bytes in length.

')' expected

expecting target : dependencies
Could not decipher target line.

ambiguous implicit rule
Two implicit rules were found for the same from and to extensions.

bad syntax for implicit rule

Implicit rule syntax is of the form: .from.to : No dependencies are allowed for an implicit rule.

undefined macro 'xxx'

don't know how to make 'xxx'

The file xxx does not exist and no rules were specified for making it.

'xxx' not found

Could not find the command xxx along the search path.

Global Optimizer Error Messages

duplicate filenames' <filename>'
Cannot open two files at once with the same name.

error opening input file' <filename>
The specified file count could not be opened.

error opening output file' <filename>
The specified output file could not be opened.

error writing output file
This usually caused by a full disk.

Global Optimizer Bug: <NUMB>
The optimizer has encountered an internal bug. Please contact ZORTECH

Intermediate file error???
The optimizer has encountered an error in the intermediate file. This is a bug in the compiler. Please contact ZORTECH.

no input file specified
A file is required on the command line.

out of memory
The optimizer has run out of memory! Large amounts of memory are needed to keep details about each function.The function being compiled can be split in two functions or just not optimized. The optimizer will use as much memory as is available to it. If you do not have a full 640K bytes of memory then getting more memory will help!

undefined flag' <IDENT>
A flag was specified on the command line that the optimizer did not recognize. Check the manual and the command line and correct it.

Warning: variable <IDENT> used before set
The optimizer determines if each variable is initialized before it is used. The variable listed was not, and may cause your program to have inexplicable results.

Runtime Error Messages

Runtime errors occur when a program that has already been compiled and linked is run. These messages are generated by the C.OBJ startup module.

Stack Overflow

The stack of the running program grew to the heap, or vice versa, and now they overlap. The problem can only be fixed by having more stack memory. This message is only generated when the program was compiled with the -s option, or the function _chkstack() was called.

Floating Point Not Loaded

The program attempted to perform floating point operations, but the floating point run-time system parts have not been linked in.

Not Enough Memory

Not enough memory is available to run the program. Use the MS-DOS CHKDSK command to see how much memory there is.

Bad Stack Size Parameter

The '=nnnn' command line parameter to set the runtime stack size was not valid.

DOS 1.xx Not Supported

Use MS-DOS 2.0 or later.

APPENDIX THREE

Converting from Previous Versions of the Zortech C Compiler

Common Blocks

Common blocks are now generated for global data with no initializer. A more advanced linker is needed to support this, use the one supplied or the one that comes with MASM 5.0 or later. Common blocks mean that if there is:

File A	File B
int var;	**int var;**

then storage for only one of the var's is allocated. Previous compilers would generate storage for both, causing errors when linking.

Chars

Characters now default to being signed, instead of unsigned. The -J switch can be used to make chars default to being unsigned.

Memory Models

Now follows naming conventions that are in more common usage:

Old	New	
C	T	Tiny (for .COM files)
S	S	Small
P	M	Medium
D	C	Compact
L	L	Large

Predefined Macros

ANSI C predefined macros __DATE__ and __TIME__ are now supported. If ZTCPP1 is used,

```
#define c_plusplus 1
```

is done. This is compatible with the AT&T C++ translator. This macro can be used to #ifdef code that needs to compile with both C and C++. The macros I8086S..I8086L are no longer generated. Instead, the macros M_I86SM, M_I86MM, M_I86CM, M_I86LM are generated. Use the macro __ZTC__ instead of the older DLC or ZC macros.

Preprocessor

Macro argument substitution inside strings and character constants is no longer done. Instead, the ANSI method of preceding the argument with a # will 'stringize' it. For example:

```
#define print(fmt,arg)
printf("format %" #fmt "\n",arg);
```

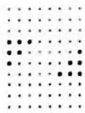
instead of the older way:

```
#define print(fmt,arg)
printf("format %fmt\n",arg);
```

Token concatenation wasn't possible with the older compiler. The new one supports the ANSI method using the ## operator.

Far Pointer Arithmetic

What happens when far pointers are cast to and from longs is now completely different. See the chapter entitled Advanced Features.



Changed Library Functions

Modified routines in DISP package to use the attribute set by disp_setattr() instead of the default white-on-black.

For stdio.h, min() and max() macros are removed. They caused too many people problems. SEEK_SET, SEEK_CUR and SEEK_END added.

bdos() and bdosx() now set errno and _doserrno if the carry flag was set upon return. They also return AX, instead of AL.

int86(), int86x(), intdos() and intdosx() now set errno and _doserrno if the carry flag was set upon return. They also set _cflag in outregs to !=0 if the carry flag was set. They also return AX instead of the value of the flags.

MACROS.ASM modified to put out MS segment names.

BR macro changed to JMPS, which is more commonly used.

Added sound.h, process.h, conio.h, direct.h, int.h, bios.h.

All time functions have had their zero moved from 1980 to 1970, to be compatible with unix times. The difference between the unix 0 and the DOS 0 is TIMEOFFSET, defined in time.h.

Linking

How files are linked is completely different. Refer to the chapter entitled Linking Programs. The method for creating integer-only executables is also completely different.

Assembly Language Interface

Now follows Microsoft conventions (except for how floats and doubles are returned). All global identifiers have an _ prepended to them.

Return values:

AX	near pointers, ints, unsigneds, shorts, chars
DX,AX	longs, far pointers, floats (DX is most significant word)
AX,BX,CX,DX	doubles

For structure returns:

AX	pointer to struct for T, S, M models
DX,AX	pointer to struct for C, L models

Registers SI, DI and DS are preserved across function calls. The direction flag is assumed to be always forward. Any function can now modify ES, in any memory model. The segment naming conventions now follow the Microsoft format. Modules compiled with Zortech C, Turbo C and Microsoft C are not interchangable. The runtime libraries are not interchangable. It is possible, though, to make a library that will successfully link with applications compiled by each compiler, but this is a task for the expert only. The reasons why they are incompatible are:

- *Different layouts of _iob struct.*
- *Different names and calling conventions for compiler support functions like long multiply.*
- *Totally different methods of doing floating point. (Zortech floating point is fully reentrant!)*
- *Variations in how structs are returned.*
- *Other compilers may generate code that assumes that DS==SS, which is not the case with Zortech C in the C and L models.*

APPENDIX FOUR

Complete ASCII Chart

ASCII value	Character	Control character		ASCII value	Character
000	(null)	NUL		032	(space)
001	☺	SOH		033	!
002	●	STX		034	"
003	♥	ETX		035	#
004	♦	EOT		036	$
005	♣	ENQ		037	%
006	♠	ACK		038	&
007	(beep)	BEL		039	'
008	(backspace)	BS		040	(
009	(tab)	HT		041	)
010	(line feed)	LF		042	*
011	(home)	VT		043	+
012	(form feed)	FF		044	'
013	(carriage return)	CR		045	-
014	♫	SO		046	.
015	☼	SI		047	/
016	►	DLE		048	0
017	◄	DC1		049	1
018	↕	DC2		050	2
019	!!	DC3		051	3
020	¶	DC4		052	4
021	§	NAK		053	5
022	▬	SYN		054	6
023	↨	ETB		055	7
024	↑	CAN		056	8
025	↓	EM		057	9
026	→	SUB		058	:
027	←	ESC		059	;
028	(cursor right)	FS		060	<
029	(cursor left)	GS		061	=
030	(cursor up)	RS		062	>
031	(cursor down)	US		063	?

ASCII value	Character		ASCII value	Character
064	@		103	g
065	A		104	h
066	B		105	i
067	C		106	j
068	D		107	k
069	E		108	l
070	F		109	m
071	G		110	n
072	H		111	o
073	I		112	p
074	J		113	q
075	K		114	r
076	L		115	s
077	M		116	t
078	N		117	u
079	O		118	v
080	P		119	w
081	Q		120	x
082	R		121	y
083	S		122	z
084	T		123	{
085	U		124	:
086	V		125	}
087	W		126	~
088	X		127	⌂
089	Y		128	Ç
090	Z		129	ü
091	[		130	é
092	\		131	â
093	]		132	ä
094	^		133	à
095	_		134	°a
096	,		135	ç
097	a		136	ê
098	b		137	ë
099	c		138	è
100	d		139	ï
101	e		140	î/
102	f		141	ì

ASCII value	Character	ASCII value	Character
142	Ä	181	╡
143	Å	182	╢
144	È	183	╖
145	Œ	184	╕
146	Æ	185	╣
147	ô	186	║
148	ö	187	╗
149	ò	188	╝
150	û	189	╜
151	ù	190	╛
152	ÿ	191	┐
153	Ö	192	└
154	Ü	193	┴
155	¢	194	┬
156	£	195	├
157	¥	196	─
158	Pt	197	┼
159	*f*	198	╞
160	á	199	╟
161	í	200	╚
162	ó	201	╔
163	ú	202	╩
164	n~	203	╦
165	Ñ	204	╠
166	a	205	═
167	o	206	╬
168	¿	207	╧
169	⌐	208	╨
170	¬	209	╤
171	½	210	╥
172	¼	211	╙
173	i	212	╘
174	«	213	╒
175	»	214	╓
176	░	215	╫
177	▒	216	╪
178	▓	217	┘
179	│	218	┌
180	┤	219	■

ASCII value	Character	ASCII value	Character
220	▄	238	ξ
221	▌	239	∩
222	▐	240	≡
223	▀	241	±
224	α	242	≥
225	β	243	≤
226	Γ	244	⌠
227	π	245	⌡
228	Σ	246	÷
229	σ	247	≈
230	μ	248	°
231	τ	249	∙
232	Φ	250	·
233	Θ	251	√
234	Ω	252	ⁿ
235	δ	253	²
236	∞	254	■
237	∅	255	(blank 'FF')

Index

#define, 259
#include files, 136, 139
& operator, 30
.COM file, 131
.EXE file, 131
.HPP files, 17
.LST file, 137
.MAP file, 144
.OBJ file, 131
80287, 135
8086 processor, 199
8087, 135,250
8087 opcodes, 242
:: operator, 21, 56
< operator, 18
_8087, 250
_assert, 497
_chkstack, 249, 288,
_ctype, 259
_farptr_norm, 316
_okbigbuf, 248
_osmajor, 255
_osminor, 256
_psp, 256
_stack, 249

A

abort, 270
abs, 271
abstract data typing,14,19, 52
access function, 41,47
acos, 272
Allocate memory, 255
allocating heap, 248
ANSI, 134, 206, 245
ANSI C standard, 16
ASCII, 259
asctime, 273
Asian language characters, 136
asin, 275
assembly language, 225, 235
assert, 276
assert.h, 259
assignment, 49
atan, 277
atof, 278
atoi, 278
atol, 278
AT&T C++, 167
AUTOEXEC.BAT, 9,60
Autoprototyping, 16, 138, 164
auxilliary port, 18

B

base class, 14, 31, 32
base pointer, 232
batch files, 24
bdos, 279
begcode, 231, 234
begdata, 231, 234
Bell Laboratories, 15
BIN, 59
binary mode, 246
binary tree, 45
bioskey, 280
bsearch, 282
bunch, 143, 170, 178

C

C Primer Plus, 13
C.ASM, 183

c_endp, 234
callm, 234
calloc, 248, 284
caux, 18
ceil, 286
CFLAGS, 62, 145
CGA, 483
char, 150
character constants, 149
chdir, 287
child process, 248
Chinese, 136
cin, 18
class, 14, 17, 19, 37
class declarations, 20
class definitions, 20
clearerr, 289
clock, 290
close, 291
code segment, 231
CodeView, 134, 135, 144, 172, 210
comments, 38
commonality, 14, 52
Compatibility, 166
compiler, 131
complex programs, 142
CONFIG.SYS, 10, 60
configuring, 63
conio.h, 259
const, 37, 151
Constant Propagation, 141, 188
constructor, 21, 24, 25, 32, 34, 43, 44
Copy proagation, 141, 188
copy-initializer, 25, 44, 47
cos, 292
cout, 18
cprn, 18
creat, 293
ctime, 295
ctype.h, 259
cursor, 235

D

data alignment, 233
data segment, 206, 231, 248
de-referencing, 45
dead assignment, 141
dead assignment elimination, 188
dead code, 141
dead variable, 141
dead variable elimination, 188
debug, 135
debuggers, 209
declaration, 16
default character type, 137
definitions, 16
delete, 42, 71
destructor, 43, 44, 148
DFA, 185
difftime, 296
direct.h, 260
disp.h, 260
Display Package, 297
div, 303
DOS Package, 304
dos.h, 261
dos_abs_disk_read, 304
dos_abs_disk_write, 304
dos_creat, 304
dos_getdiskfreespace, 304
dos_set_cntl_break, 304
dos_set_verify, 304
DOSSEG, 242
double, 150

E

ecvt, 307
EGA, 483
_8087, 250
endcode, 231, 234

enddata, 231, 234
Environment Variables, 60, 145
errno, 256
errno.h, 264
error limit, 138
exe2bin, 131, 143, 177
execl, 308
execvp, 308
exit, 56, 310
exp, 312
EXTRN, 241

F

fabs, 313
far calls, 200
far pointer, 150, 199, 201, 207
far pointer, normalized, 203
farcalloc, 314
fclose, 318
fcvt, 319
feof, 320
ferror, 321
fflush, 322
FG59, 483
fg.h, 262
fg_activepage, 495
fg_adjustxy, 499
fg_blit, 501
fg_box_cpy, 502
fg_charbox, 492
fg_display, 494
fg_displaybox, 491
fg_displaypage, 495
fg_drawarc, 503
fg_drawbox, 504
fg_drawdot, 505
fg_drawellipse, 506
fg_drawline, 507
fg_drawlineclip, 508
fg_drawmatrix, 509
fg_drawthickline, 511
fg_fillbox, 512
fg_flush, 513
fg_getcolormap, 514
fg_init, 515
fg_init_null , 516
fg_line_cpy, 518
fg_lineclip, 498
FG_MODE_SET, 488
FG_MODE_XOR, 488
fg_ncolormap, 492
fg_nsimulcolor, 493
fg_numpages, 494
fg_pixelx, 493
fg_putc, 519
fg_puts, 520
fg_readbox, 521
fg_readdot, 523
fg_restore, 524
fg_save, 525
fg_setactivepage, 526
fg_setcolormap, 527
fg_setdisplaypage, 528
fg_setlinepattern, 529
fg_term, 530
fg_writebox, 531
fgetc, 323
fgets, 324
file handling, 245
file manipulation, 18
File mode, 246
fileno, 325
filesize, 326
findfirst, 327
fixup errors, 241
float, 150
floating point, 135, 155, 250
floor, 329
flushall, 330
fmod, 331
fopen, 332
FP_OFF, 334
FP_SEG, 334
fprintf, 335
fputc, 336
fputs, 337
fread, 338

free, 42, 248, 339
free store, 42, 45
freopen, 340
frexp, 341
friend class, 55
friend function, 26, 41, 55
fscanf, 342
fseek, 343
fstat, 345
ftell, 347
func, 234
function overloading, 24
function parameters, 46
Function Prototyping, 16
fwrite, 348

G

garbage, 49
getc, 349
getcwd, 351
getDS, 352
getenv, 353
gets, 354
global common subexpressions, 141, 188
global data, 231
global optimizer, 140, 144
global variables, 56, 250

H

hard disk, 59
header files, 17, 23, 38, 259, 261, 263, 265, 267
heap, 42, 49, 248
heap allocation, 56
Hercules, 483
hex, 149
hypot, 355

I

Identifier conflicts, 147
identifiers, 148
Implicit Rules, 220
INCLUDE, 59, 136, 145, 158
index, 356
inheritance, 14, 31, 52
inherited class, 32
Initializers, 154
inline, 41
inline functions, 33, 34, 40, 357, 398
inp, 357
input file, 142
inpw, 357
Insert, 71
int, 150
int.h, 264
int86, 362
intdos, 365
integer only programs, 144, 180
Intel, 131
internal stack, 142
Interrupt Package, 359
io.h, 264
is package, 368
isatty, 367
itoa, 371

J

Japanese, 136

K

kbhit, 373
Kernighan & Ritchie, 245
Korean, 136

L

labs, 374
LCODE, 234
ldexp, 375
ldiv, 376
LIB, 59, 145
library routines, 131
limits.h, 264
line, 209
linker, 131
linker response file, 144
linking, 173, 175, 177, 179
listing file, 137
local variable, 48, 232
localtime, 378
log, 380
long, 150
longjmp, 381
loop, 141, 188
Loop induction, 141
Loop induction variables, 188
Loop invariant removal, 141, 188
LPTR, 234
lseek, 383

M

Macros, 157, 217, 231, 234, 242
main, 56
make, 24, 211
make aArguments, 216
MAKEFILE, 212
malloc, 42, 248, 385
map file, 148
MASM, 143, 220, 242
math.h, 258, 264
matherr, 386
member function, 20, 21, 26, 31, 37, 41, 49
member object, 34, 55
memchr, 388
Memory Allocation, 42, 248
memory model, 138, 144, 199, 200, 231, 238, 242
messages, 14, 52
Microsoft CodeView, 210
Microsoft Link, 134, 171
MK_FP, 205, 391
mkdir, 390
mktime, 392
modf, 393
module constructors, 171
module destructors, 171
Mouse Package, 394
MS-LINK, 171
msmouse.h, 265
multiple object assignment, 49

N

NDEBUG, 259
near or far keywords, 206
Near pointers, 150, 201, 206
new, 42, 56
NULL pointer, 209
numeric data processor, 250

O

object, 14, 19, 37
object modules, 23

Object-Oriented Programming, 14, 15, 24
objtoasm, 224, 235
octal, 149
offset, 199
open, 396
operator, 28,48
operator overloading, 18, 32, 28, 49
operator+, 30
operator++, 30
operator=, 30, 49
optimizer, 185, 187, 189, 191, 193, 195, 197
outp, 398
output file, 138, 140, 142
overflow errors, 183

P

P, 234
Pascal, 16
pass-by-reference, 30, 47
PATH, 60
peek, 400
perror, 402
Pointers, 154
poke, 403
poly, 405
polymorphism, 14, 52
port access, 261
pow, 406
predefined macros, 138, 157, 202, 218
preprocessor, 37, 135, 156
Preprocessor Directives, 159
printer, 18
printf, 18, 407
private, 19, 41, 55
process.h, 265
Program Segment Prefix, 229
protected, 55
Prototype Induced Cast, 163
Prototyping, 16, 161
pseudo-ops, 231
public, 19, 31, 55
putc, 412
puts, 413

Q

qsort, 414
Quick Start, 8

R

raise, 416
rand, 418
read, 419
realloc, 248, 421
reference, 30, 42, 44, 45
reference counting, 49
register assignments, 141
register usage, 233
registers, 199
rename, 423
reserved words, 147
Response Files, 221
return values, 233
rewind, 424
rmdir, 425
ROM, 151
runtime library, 137

S

sbrk, 426
scanf, 18, 428
scope resolutionoperator, 56
scoping, 37, 39, 42
search, 71

segment, 199
segment registers, 199, 432, 225
segment wrap, 204
segread, 432
setbuf, 433
setjmp, 434
setjmp.h, 265
setvbuf, 435
signal, 438
signal.h, 265
sin, 440
SIZEPTR, 234
sound , 441
sound.h, 265
source file, 131
space, 188
space optimizations, 141
spawn, 248, 443
sprintf, 445
SPTR, 234
sqrt, 446
srand, 447
sscanf, 448
stack, 42, 44, 249
stack frame, 232
stack overflow, 134, 142, 249
stack pointer, 42, 232
stack size, 144, 183
standard output, 18
startup module, 248
stat, 449
stat.h, 267
static constructor, 148
static data, 231
static variables, 56, 148
stdarg.h, 266
stddef.h, 266
stderr, 18
stdin, 18
stdio.h, 266
stdlib.h, 267
stdout, 18
strcat, 451
streams, 18, 28
strerror, 453
string.h, 267
strlen, 454
strlwr, 455
strncat, 456
Stroustrup, 13, 15, 18, 20, 37
strrchr, 458
strspn, 459
strstr, 460
strtod, 461
structure, 26
structure alignment, 134, 208
strupr, 464
swab, 465
switches , MASM, 242
switches, command line, 133
switches, compiler, 134
switches, ZTC, 143, 242
SYMDEB, 135
SYS, 59
sys\stat.h, 267
system, 466

T

TEMP, 61, 145
text editor, 131
text mode, 246
this, 30, 49
time, 188, 469
time optimizations, 141
time.h, 267
TMP, 61, 145
toascii, 470
Toshiba 3100 machine, 483
touch, 223
type checking, 16
Type Conversion, 167
type derivation, 14
Types, 150

U

ungetc, 472
UNIX, 143
unlink, 474
unsigned, int 150
unsigned char, 150
unsigned long, 150
UTIL, 59
utime, 475

V

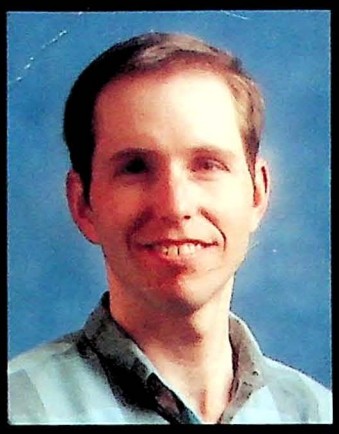
va_arg, 477
vbe, 188
vector class, 47
verbose compile, 138, 140
verbose mode, 142
very busy expressions, 141
VGA, 483
virtual functions, 14, 52, 54
void pointers, 51
volatile, 151
vprintf, 478

W

warning messages, 138
Weiner and Pinson, 13, 18, 37,54
wildcard expansion, 181
word processor, 131
Wordstar, 74
write, 480

Z

ZCHCONF, 63
ZED, 67
ZED buffers, 69
Zorlib, 535
Zortech, 13,199
Zortech C, 16
Zortech C++, 16, 35, 37
Zortech C++ Newsletter, 35
__ZTC__, 157
ZTC, 131, 143, 242
ZTC1, 133, 140
ZTC1B, 133, 140
ZTC2, 131, 140, 142
ZTC2B, 140, 142
ZTCHELP, 63
ZTCPP1, 131, 133, 140
ZTCSETUP, 59, 67
ZTG, 140



Walter Bright

About The Compiler Author . . .

Walter received a BSME in 1979 from Caltech at age 20. While writing the popular Empire game, he became frustrated with slow C compilers which ignored well-known optimization techniques. He decided to write his own so he could maintain a bug-free compiler with the speed and features he wanted for his own projects.

In 1983, Walter created Datalight C, which was 3 times faster in compilation speed, and equal or better in execution time than Lattice C (the industry leader at the time).

Using techniques he learned when writing the Datalight & Zortech C compilers, he tackled the formidable task of building a native-code C++ compiler (instead of a slow and tedious translator, which requires you to buy someone else's C compiler). He is committed to keeping Zortech C++ the fastest, most efficient and easiest-to-use compiler in the industry.

Walter lives in Seattle, Washington.

Bruce Eckel, Micro Cornucopia Magazine

ZORTECH C++ COMPILER

Zortech C++ is the first native-code C++ compiler for MS-DOS compatible machines. This means you don't need to spend a lot of money on a C compiler. In fact, you now hold in your hands a C++ compiler, a C compiler, a linker, a librarian, a fully integrated editing environment, context sensitive help and the fastest graphics library you've ever seen!

With Zortech C++, you can mix and match code in C++, C and assembler and link them all together with a single command!

Zortech C++ comes with a complete C++ and C library, including compatibility with the ANSI C draft. Unique to Zortech C++ is support for mice, fast screen writes, re-entrant floating point and speaker support.

Now you can move your Microsoft and Turbo C code over to Zortech C++ – we've included compatible library functions! Also, we have made Zortech C++ compatible with Microsoft's Codeview debugger!

Enter the world of Object Oriented Programming (oops) with Zortech C++!

Sure, C is great – but C++ is better!

Zortech, Inc., 366 Massachusetts Avenue, Arlington, MA 02174
(617)-646-6703 Fax: (617)/643-7969

Zortech Ltd., 106-108 Powis Street, London SE18 6LU, England
01-316-7777 Fax: 01-316-4138

Performance benchmarks

Benchmark	Zortech C	Zortech C++	Turbo C 1.5	Quick C 1.0
Sieve	20.49	20.54	23.62	22.72
Rsieve	20.49	20.54	23.62	22.03
Integer	1.32	1.38	6.31	6.49
Float	0.17	0.22	52.29	51.03
*Float	32.73	37.74	52.39	51.63
Pointer	17.91	17.96	17.13	16.87
Rpointer	17.79	17.91	17.14	16.64
Loop	3.90	3.90	3.90	3.90
Optimize	0.49	0.60	8.46	8.79

Benchmarks were run on an 80286 based IBM compatible at 6Mhz with no 8087.

*The float benchmark was re-run without optimization, due to the Zortech optimizer determining that the code was unused and removing it.

NOTE: Zortech C is included in the Zortech C++ package.